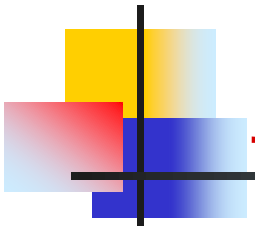


主要内容

- 回溯法的算法框架
- 装载问题
- *n*后问题
- 0-1背包问题
- 图的*m*着色问题
- 旅行售货员问题
- 连续邮资问题
- 回溯法效率分析

回溯法的算法框架(1)

- 回溯法(BackTracking)：“通用的解题法”

- 回溯法的按深度优先策略：

1）首先定义问题的解空间。

2）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(2)

- 回溯法的按深度优先策略示例1

- 回顾0-1背包问题(0-1 Knapsack Problem)：给定n种物品和一背包。物品i的重量是w_i，价值为v_i，背包的容量为c。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?

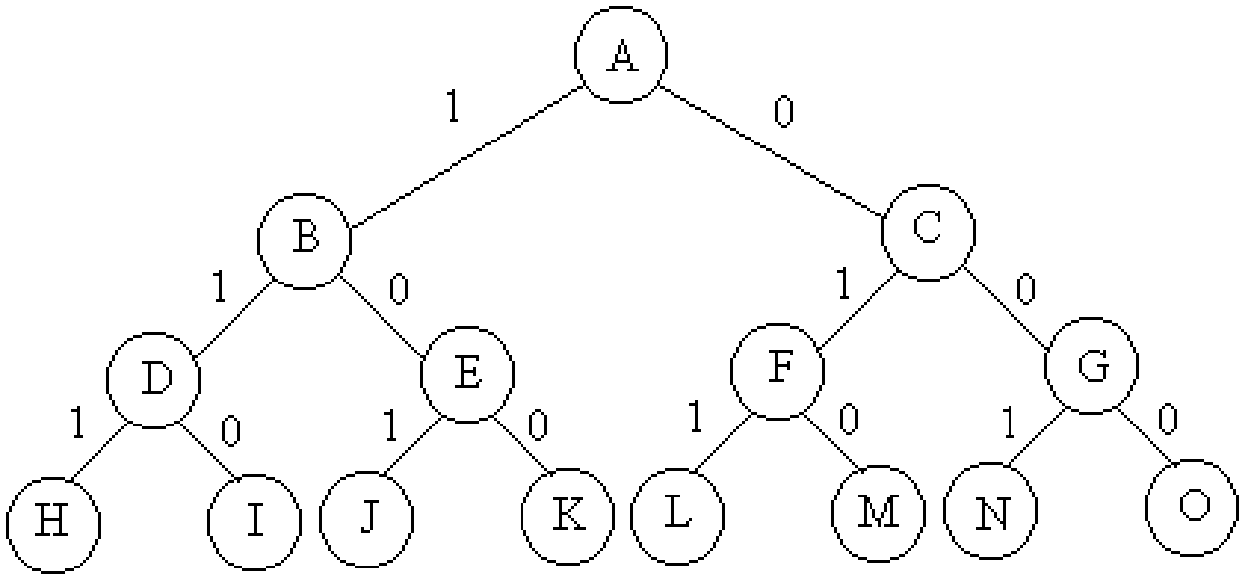
- 形式化描述如下：给定$c>0, w_i>0, v_i>0, 1\le i\le n$，请找出$n$元0-1向量$(x_1, x_2,\ldots, x_n)$，$x_i \in \{0,1\}$，$1\le i\le n$，使得：

规划目标：$\max \sum_{i=1}^{n} v_i x_i$　约束条件：$\begin{cases} \sum_{i=1}^{n} w_i x_i \le c \\ x_i \in \{0,1\}, 1 \le i \le n \end{cases}$

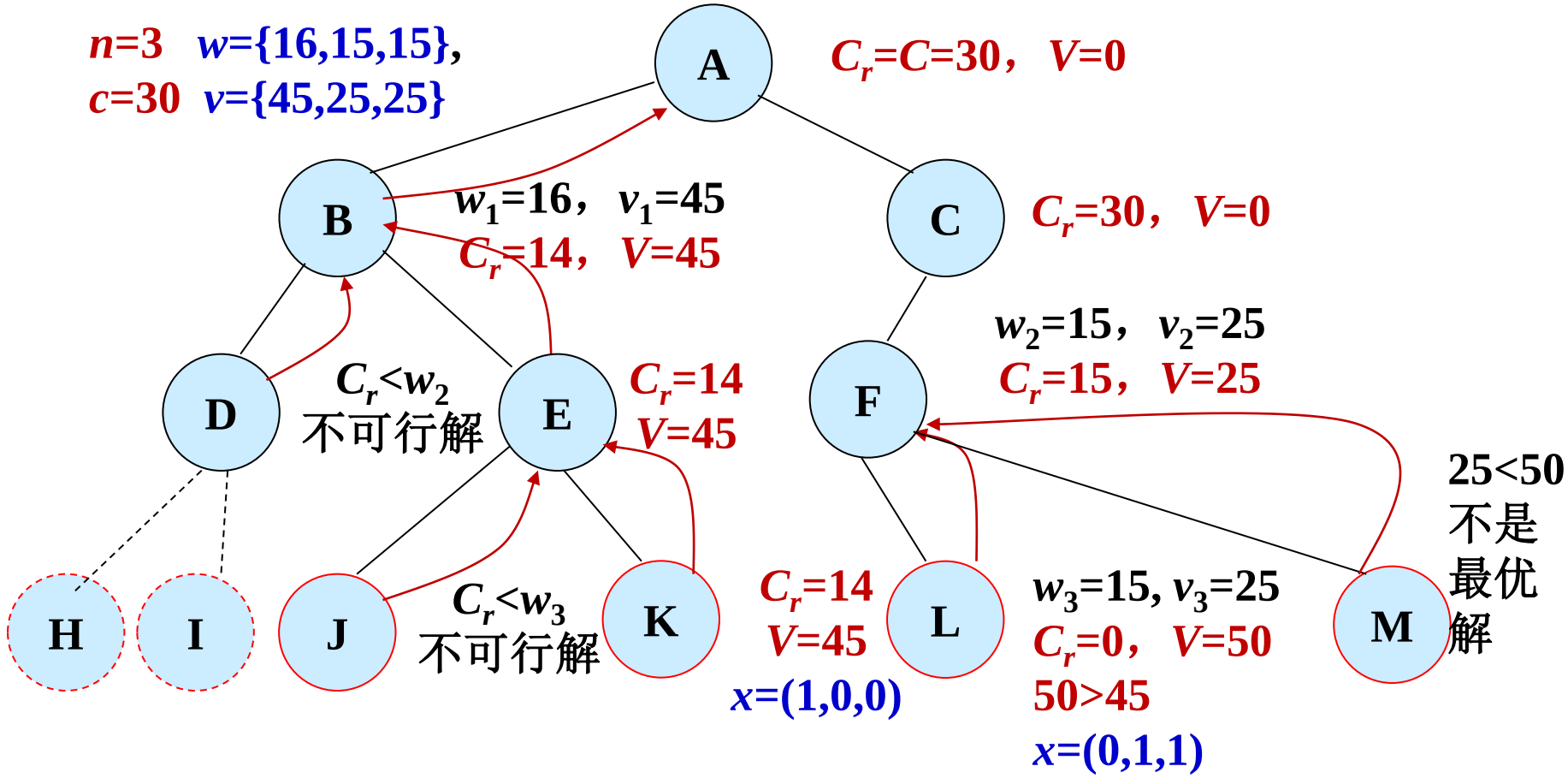
回溯法的算法框架(3)

- 令：$n=3$, $c=30$, $w=\{16,15,15\}$, $v=\{45,25,25\}$
- 显然当$n=3$时，0-1背包的解空间为(共2^3个解)：

物品1	物品2	物品3
1	1	1
1	1	0
1	0	1
1	0	0
0	1	1
0	1	0
0	0	1
0	0	0



回溯法的算法框架(4)



深度优先策略搜索$n=3$, $c=30$, $w=\{16,16,15\}$, $v=\{45,25,25\}$ 的解空间

回溯法的算法框架(5)

引入相关概念：

- 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）。

回溯法的算法框架(6)

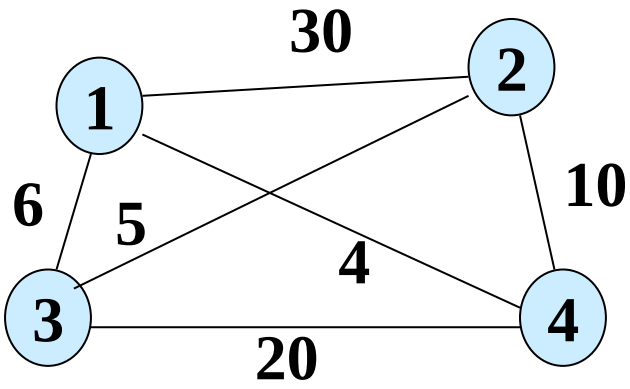
- 回溯法的按深度优先策略示例2：

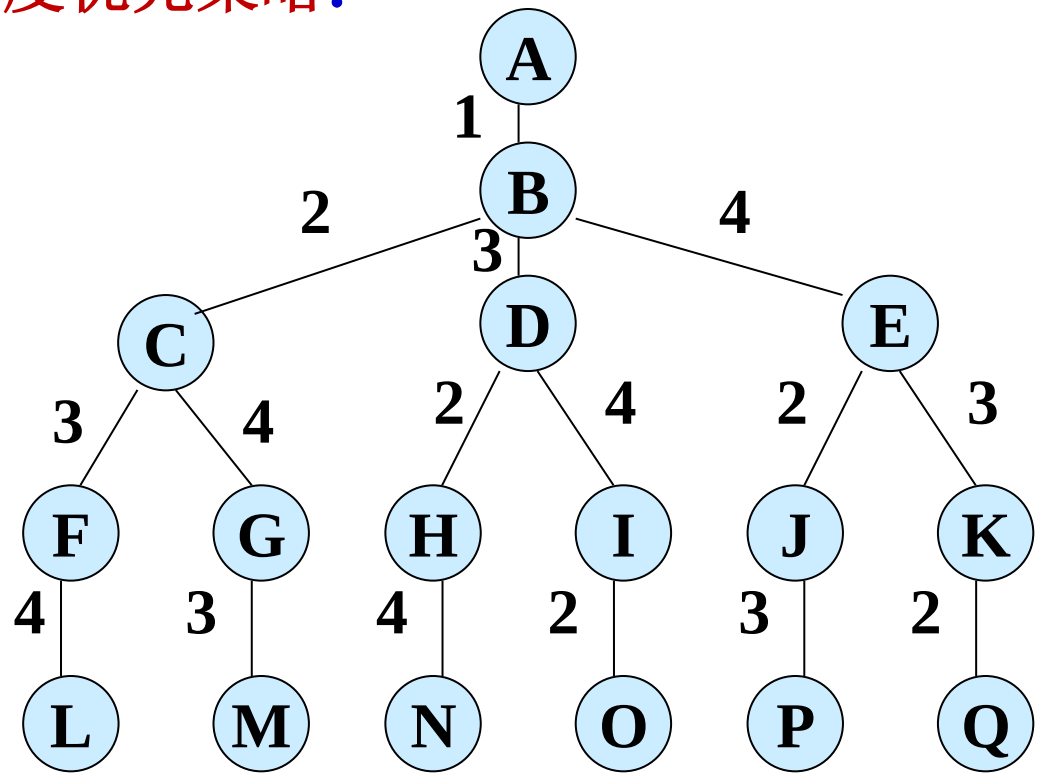
- 旅行售货员(Traveling Salesman Problem)问题：某售货员要到若干城市去推销商品，已知各城市之间的路程，他要选定一条从驻地出发，经过每个城市一遍，最后回到住地的路线，使总的旅费最小。

- 上述问题可转化为无向带权全连通图的遍历问题，这是NP完全问题：如果有*n*个顶点(城市)，则最多可能有(*n*-1)!条路线。

回溯法的算法框架(7)

- 旅行售货员问题的深度优先策略：





问题最优解：(1,3,2,4,1)，最优值25

回溯法的算法框架(8)

- 回溯法的深度优先策略的优化：使用剪枝法来避免无效搜索。这类称为剪枝函数，通常剪枝有两种策略：

1) 用约束函数在扩展结点处减去不满足约束的子树。如：

0-1背包问题中可剪去导致不可行解的子树。

2) 用限界函数剪去得不到最优解的子树。

旅行员背包问题中可剪去费用已经超过现有最好的周游路线费用。

回溯法的算法框架(9)

- **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);
        }
}
```

1) t表示递归深度，n表示最大递归深度;
2) output(x)表示输出可行解x; h(i)表示在当前扩展结点x[t]的第i个可选值。
3) f(n,t)和g(n,t)分别表示在当前扩展结点处未搜索过的子树的起始编号和终止编号。
4) contraint(t)和Bound(t)表示在当前扩展结点处的约束函数和上界函数。

回溯法的算法框架(10)

- 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--;
  }
}
```

说明：

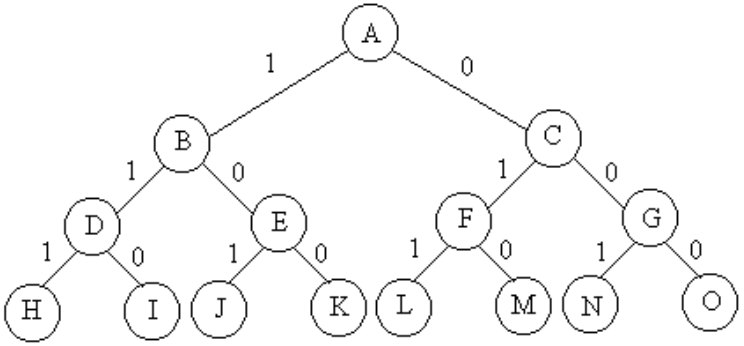
1) solution(t)判断在当前扩展结点处是否已经得到问题的解;如已得到，解为：x[1:t]。

2) f(n,t)和g(n,t)分别表示在当前扩展结点处未搜索过的子树的起始编号和终止编号。h(i)表示在当前扩展结点x[t]的第i个可选值。

3) contraint(t)和Bound(t)表示在当前扩展结点处的约束函数和上界函数。

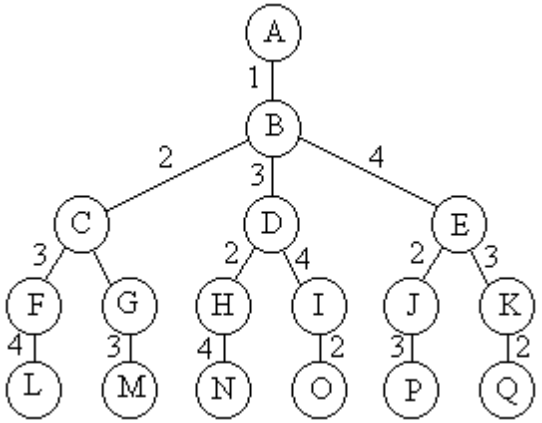
回溯法的算法框架(11)

- 典型的解空间树：子集树和排列树。



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);
     }
}
```



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]);
     }
}
```

主要内容

- 回溯法的算法框架
- 装载问题
- *n*后问题
- 0-1背包问题
- 图的*m*着色问题
- 旅行售货员问题
- 连续邮资问题
- 回溯法效率分析

装载问题(1)

- 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：$\sum_{i=1}^{n} w_i \le c_1 + c_2$

- 求解目标：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。示例：

$n=3, c_1=c_2=50, w=\{10,40,40\}$ ⟶ 问题有解

$n=3, c_1=c_2=50, w=\{20,40,40\}$ ⟶ 问题无解

装载问题(2)

- **装载问题：**

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(可证明)：

(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。此时装载问题等价于以下特殊的**0-1**背包问题：

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

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

装载问题(3)

- 装载问题的回溯法实现-主类Loading：

```
template<class T>
class Loading {
   friend MaxLoading(T[], T, int);
   private:
   void BackTrack(int i); //表示搜索第i层子树
   int n; // 货箱数目
   T *w, // 货箱重量数组
         c, // 第一艘船的容量
         cw, // 当前装载的重量
         bestw; // 目前最优装载的重量
};
```

装载问题(4)

- 装载问题的回溯法实现-核心函数**BackTrack**：

```
template<class T>
void Loading<T>::BackTrack(int i){//从第i 层节点搜索
    if (i > n) {//位于叶节点
        if (cw > bestw) bestw = cw; return;
    }
    //检查子树
    if (cw + w[i] <= c) {// 尝试x[i] = 1
      cw += w[i];
      BackTrack(i+1) ;
      cw -= w[i];
    }
    BackTrack(i+1);// 尝试x[i] = 0
}
```

算法的复杂性为：$O(2^n)$

装载问题(5)

- 装载问题的回溯法实现-初始函数**MaxLoading**：

```
template<class T>
T MaxLoading(T w[], T c, int n){// 返回最优装载的重量
  Loading<T> X;
  //初始化X
  X.w = w;
  X.c = c;
  X.n = n;
  X.bestw = 0;
  X.cw = 0;
  // 计算最优装载的重量
  X.BackTrack(1); //从第1层开始搜索
  return X.bestw;
}
```

装载问题(6)

- **MaxLoading初始函数执行示例：**

  n=4, c_1=40, c_2=40, w={ 35, 15, 25, 5}，解空间：2^4

c=40
n=4
i=1
bestw=0
cw=0

x[1]	1	1	1	1	1	0	0	0	0	0	0	0	0
x[2]	**1**	0	0	0	0	1	1	1	1	0	0	0	0
x[3]			**1**	0	0	1	1	0	0	1	1	0	0
x[4]				1	0	**1**	0	1	0	1	0	1	0

装载问题(7)

- 定义上界函数进行优化：

1) 引入成员r，r是剩余集装箱的重量：$r = \sum_{j=i+1}^{n} w_j$

2) 定义上界函数bound=cw+r，其中cw是当前装载重量；

3) 假设Z是解空间树第i层上的当前扩展结点，则以Z为根结点的子树中任一叶节点所相应的载重量均不超过cw+r。

故：当cw+r<=bestw时(其中bestw是当前最优载重量)，可将Z的右子树剪去。

装载问题(8)

- 优化的核心函数**BackTrack(P122)**：

```
template<class T>
void Loading<T>::BackTrack(int i){//从第i 层节点搜索
    if (i > n) {//位于叶节点
        if (cw > bestw) bestw = cw;
        return;
    }
    //检查子树
    r - = w[i];
    if (cw + w[i] <= c) {//尝试x[i] = 1
        cw += w[i];
        BackTrack( i + 1 ) ;
        cw -= w[i];
    }
    if (cw + r > bestw) //尝试x[i] = 0
        BackTrack( i + 1 ) ;
    r += w[i];
}
```

算法的复杂性依然为：$O(2^n)$

装载问题(9)

- 优化的初始函数**MaxLoading**：

```
template<class T>
T MaxLoading(T w[], T c, int n){// 返回最优装载的重量
  Loading<T> X;
  //初始化X
  X.w = w;
  X.c = c;
  X.n = n;
  X.bestw = 0;
  X.cw = 0;
  X.r=0;
  for(int i=1; i<=n; i++) X.r+=w[i];
  // 计算最优装载的重量
  X.BackTrack(1); //从第1层开始搜索
  return X.bestw;
}
```

装载问题(10)

- 含构造最优解的核心函数BackTrack：引入数组成员x和bestx。x数组用于记录当前从根至当前节点的路径，bestx数组记录当前最优解。

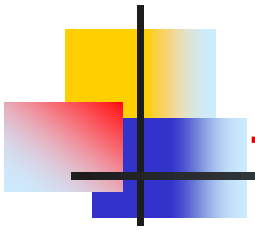
```
template<class T>
void Loading<T>::BackTrack(int i){//从第i 层节点搜索
    if (i > n) {//位于叶节点
       if (cw > bestw) {
          bestw = cw; for(j=1; j<=n; j++) bestx[j]=x[j]
          return; }
    r - = w[i]; //检查子树
    if (cw + w[i] <= c) {//尝试x[i] = 1
       cw += w[i]; x[i] = 1
       BackTrack( i + 1 ) ;
        cw -= w[i];
    }
    if (cw + r > bestw){ //尝试x[i] = 0
       BackTrack( i + 1 ) ; x[i] = 0
    }
    r += w[i];
}
```

装载问题(11)

- 迭代回溯：因为数组x 中记录可在树中移动的所有路径，故可以消除大小为n的递归栈空间。

```
template<class T>
T MaxLoading(T w[], T c, int n, int bestx[])
{ // 返回最佳装载及其值
  // 初始化根节点
  int i = 1; // 当前节点的层次
  // x[1:i-1] 是到达当前节点的路径
  int *x = new int [n+1];
  T bestw = 0, // 迄今最优装载的重量
  cw = 0, // 当前装载的重量
  r = 0; // 剩余货箱重量的和
  for (int j = 1; j <= n; j++) r += w[j];
  // 在树中搜索
  while (true) { // 尽可能下移进入左子树
     while (i <= n && cw + w[i] <= c) {
        r -= w[i]; // 移向左孩子
        cw += w[i]; x[i] = 1; i + + ;
     }
```

```
     if (i > n) {// 到达叶子
       for (int j = 1; j <= n; j++) bestx[j] = x[j];
       bestw = cw;
     }else { //移向右孩子
       r -= w[i]; x[i] = 0; i++;
     }
     // 必要时返回
     while (cw + r <= bestw) {
        i-- ; //本子树没有更好的叶子，返回
        while (i > 0 && !x[i]) {
           // 从右孩子返回
           r += w[i];  i - - ;
         }
        if (i = = 0){delete [] x; return bestw;}
        //进入右子树
           x[i] = 0; cw-= w[i]; i + + ;
        }
     }
}
```

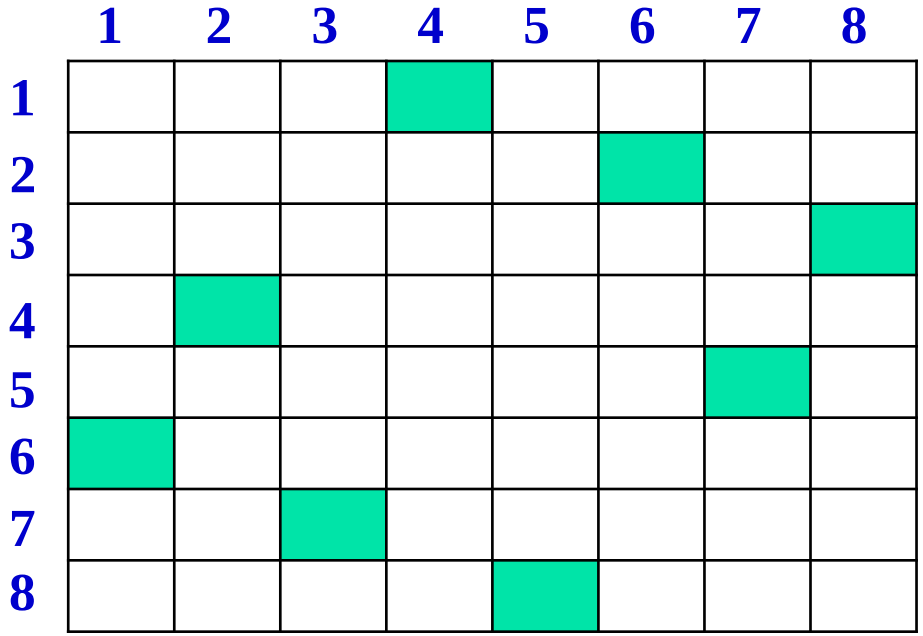


主要内容

- 回溯法的算法框架
- 装载问题
- **n后问题**
- **0-1背包问题**
- 图的*m*着色问题
- 旅行售货员问题
- 连续邮资问题
- 回溯法效率分析

n后问题(1)

- ***n*后问题(起源于1850年高斯提出的8皇后问题)**：在***n*×*n*格**的棋盘上放置彼此不受攻击的*n*个皇后。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(2)

- **n后问题求解：**

令：x[1:n]表示问题的解，其中x[i]表示皇后i放在第i行的第x[i]列。下面考虑约束条件：

1) 非同列约束：因两个皇后不能处在同一列，故各x[i]的值均不能相等。

2) 非同行约束：根据x[i]的定义可知各皇后肯定不同行。

3) 非斜线约束：考虑n×n网格的两个单元格(i,j)和(k,l)，若两个单元格同一斜线，则必有：

(i-j)=(k-l) 或 (i+j)=(k+l)，进一步有：

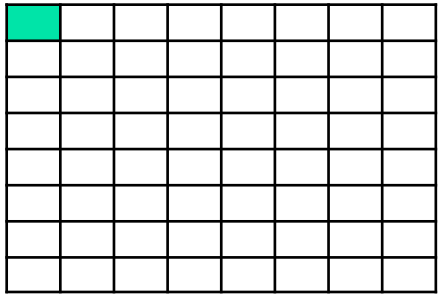
(i-k)=(j-l) 或 (i-k)=(l-j)， 因此：

若 |i-k|<>|j-l|，则单元格(i,j)和(k,l)肯定不在同一斜线。

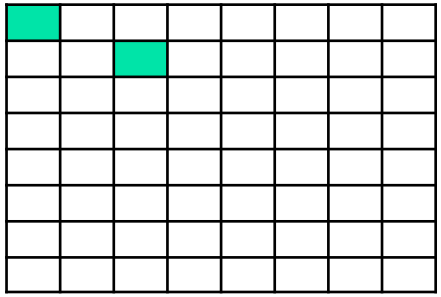
n后问题(3)

- n后问题的解空间(无约束)：n^n
- n后问题的回溯法示例(n叉树)：

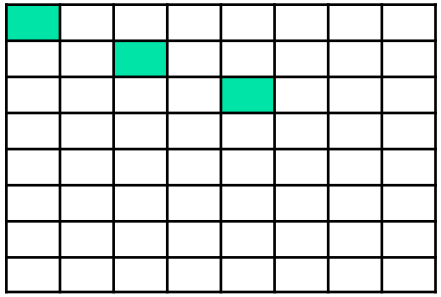
(1)



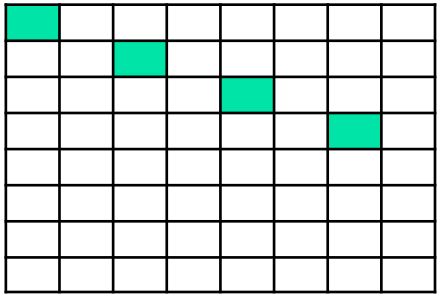
(2)



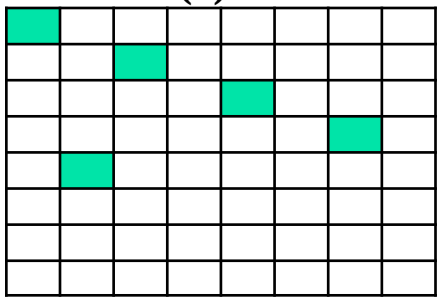
(3)



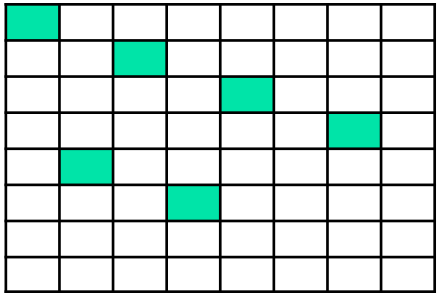
(4)



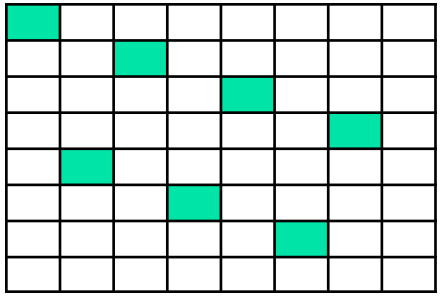
(5)



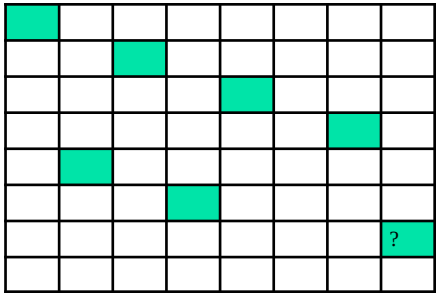
(6)



(7)



(8)



此时Queen8已无法放下，需继续回溯Queen7, Quee6…直至选取一个满足约束的位置……

n后问题(4)

- 回溯法实现-主类、约束判断和主程序：

```
Class Queen{ //主类
  friend int nQueen(int);
  private:
     bool Place(int k);  //约束函数
     void Backtrack(int t); //核心函数
     int n, //皇后数
         *x; // 当前解
     long sum; //当前已找到的可行方案
  };
```

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

```
//主程序
int nQueen(int n){
   Queen Q;
   int *p= new int[n+1];//生成结果数组
   for(int i=0; i<=n; i++){
      p[i]=0;
   }//初始化结果数组
   Q.x=p;
   Q.Backtrack(1);
   delete [] p;
   return Q.sum;
}

void main(){
    cout<<nQueen(n)<<endl;
}
```

n后问题(5)

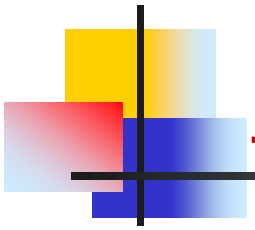
- 递归回溯法实现-核心函数BackTrack：

```
void Queen::Backtrack(int t){
   if( t>n ){
     sum++; //解方案数加1
   }else{
      for(int i=1; i<=n; i++){
         x[t]=i;
         if(Place(t))
            Backtrack(t+1);
    }
   }
}
```

n后问题(6)

- 迭代回溯法实现-核心函数BackTrack：

```
void Queen::Backtrack(void) {
  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){
       if(k==n){
          sum++;
       }else{
          k++;  x[k]=0;
       }
      }else  k--; //回溯
   }
}
```



主要内容

- 回溯法的算法框架
- 装载问题
- n后问题
- **0-1背包问题**
- 图的*m*着色问题
- 旅行售货员问题
- 连续邮资问题

0-1背包问题(1)

- **0-1背包问题的回溯实现-核心函数BackTrack。**

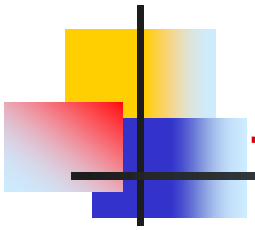
```
void Knap::BackTrack(int i)  {
    if(i>n) {
        if(bestp<cp) bestp=cp;
          return;
    }
    if(cw+w[i]<=c){//记录进入左子树
        x[i]=1;
        cw+=w[i];  cp+=p[i];
        BackTrack(i+1);
        cw-=w[i];  cp-=p[i];
    }
    if(Bound(i+1)>bestp){//记录进入右子树
        x[i]=0;
        BackTrack(i+1);
    }
}
```

算法复杂度为：$O(n2^n)$

0-1背包问题(2)

■0-1背包问题的上界函数：首先对物品按单位价值从大到小进行排序，然后按顺序装入物品，具体实现如下：

```
template<class Typew, class Typep>
Typep Knap<Typew, Typep>::Bound(int i){// 计算上界
  Typew cleft = c - cw;  // 剩余容量
  Typep b = cp;
   while (i <= n && w[i] <= cleft) {// 以物品单位重量价值递减序装入物品
     cleft -= w[i];
     b += p[i];
     i++;
  }
  // 装满背包
  if (i <= n)   b +=p[i]/w[i]*cleft;
  return b;
}
```



主要内容

- 回溯法的算法框架
- 装载问题
- n后问题
- 0-1背包问题
- 图的m着色问题
- 旅行售货员问题
- 连续邮资问题
- 回溯法效率分析

图的m着色问题(1)

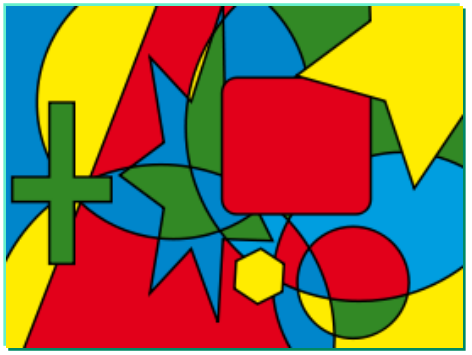
- 图的m可着色问题描述：给定无向连通图G和m种不同的颜色。用这些颜色为图G的各顶点着色，每个顶点着一种颜色，是否有着色法使G中每条边的2个顶点颜色不同？如果有，请给出所有的着色方案。

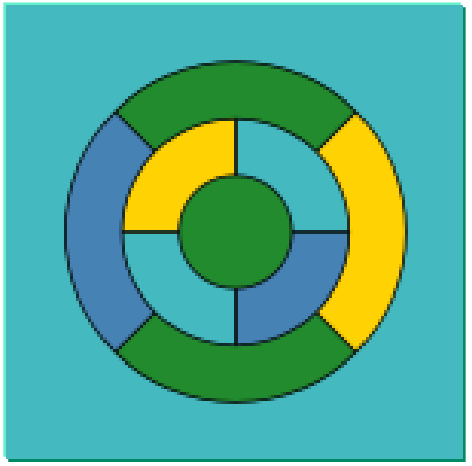
- 图的色数：若一个图最少需要m种颜色才能使图中每条边连接的2个顶点着不同颜色，则称m为该图的色数。

- 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，并使得相邻国家在地图上颜色不同。每个国家是一个单连通域，两个国家相邻指它们有一段长度不为0的公共边界。

图的m着色问题(2)

- 近代数学三大难题之一：四色猜想(Four color theorem)
  - 1852年，伦敦大学学院(University College London) De Morgan(1806.06-1871.03)的学生Francis Guthrie(1831.01-1899.10)在给英格兰地图着色时发现该问题。
  - 1879年，Bray Kempe(1849.07-1922.04)在Nature杂志上宣布自己“证明”了该猜想，并因此获得诸多荣誉，但他的假证明为后人提供了基础。
  - 1890年，John Heawood(1861.09-1955.01)指出了Kempe的证明错误(反例)，同时证明了使用5种颜色即可完成平面或球面地图着色，此后他和四色猜想纠结了近60年。
  - 1976年，Kenneth Appel(1932.10-)和Wolfgang Haken(1928.04-)总结了1936种特殊类型的地图，并借助计算机程序证明该问题，但受到部分数学家的质疑。
  - 1980年，Swart质疑这种证明非人力所能检验。2002年，Wilson再次提出质疑。

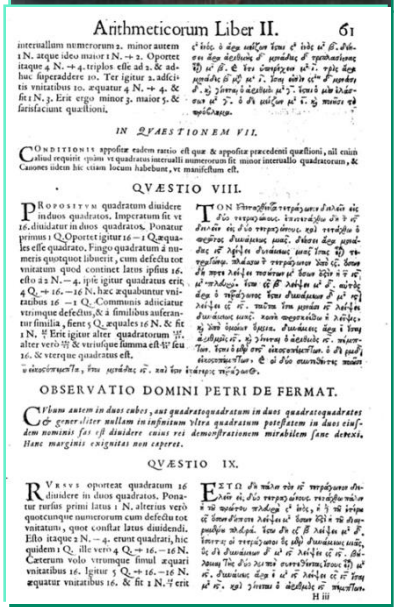




图的m着色问题(3)

- 近代数学三大难题之二：费马大定理(Fermat's Last Theorem)

> **No three positive integers *x*, *y*, and *z* can satisfy the equation $x^n + y^n = z^n$ for any integer value of *n* greater than two.**



Arithmeticorum Liber II. 61

IN QVAESTIONEM VII.

QVÆSTIO VIII.

OBSERVATIO DOMINI PETRI DE FERMAT.

QVÆSTIO IX.

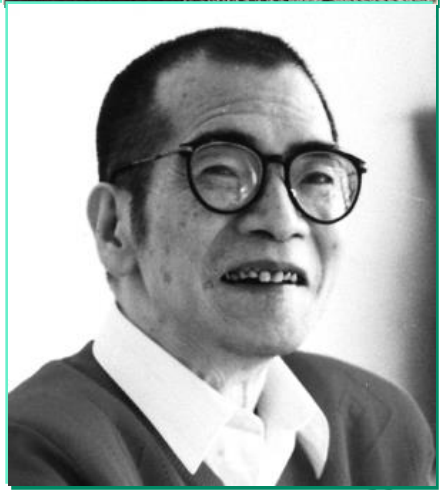
- 1637年，法国律师、业余数学王子Pierre de Fermat(1601.08-1665.01)在阅读<<算术>>时顺手写下了这个猜想(*I have discovered a truly remarkable proof which this margin is too small to contain*)。
- 1753年，瑞士数学家和物理学家 Leonhard Euler(1707.04-1783.09)宣称“证明”了n=3时的正确性，但在1770年被发现有错误推论。
- 法国女数学家Sophie Germain(1776.04-1831.06)将该猜想推论至两种情形：1) Case 1: None of *x*, *y*, *z* is divisible by *n*; 2) Case 2: One and only one of *x*, *y*, *z* is divisible by *n*.
- 1825年，德国数学家Lejeune Dirichlet证明了case2两种情形之一。
- 此后近170年，无数数学家为之奋斗，并推进着猜想的证明，并引出了Epsilon conjecture, Faltings' theorem, Taniyama-Shimura theorem(谷山-志村定理)等。
- 1995年，Andrew Wiles和Richard Taylor证明了谷山-志村定理的特例，最终完成了费马大定理的证明。

图的m着色问题(4)

- 近代数学三大难题之三：哥德巴赫猜想(Goldbach's conjecture)

> Every even integer greater than 2 can be represented as the sum of two primes.

- 1742年，德国数学家、律师Christian Goldbach(1690.03-1764.11)在写给Euler的信中提出该猜想（即：“1+1”），并随后提出了“Every odd number is the sum of three primes”的奇数哥德巴赫猜想。
- 1921年，Hardy(我国著名数学家闵嗣鹤为其学生)和Littlewood利用黎曼假设(Riemann’s Hypothesis，尚未证明)证明了哥德巴赫猜想。
- 1937年，俄罗斯数学家Vinogradov证明了奇数哥德巴赫猜想。
- 1966年，我国数学家陈景润(1933.05-1996.03)证明了“Every sufficiently large even number can be written as the sum of either two primes, or a prime and a semiprime(the product of two primes)”，即“1+2”，成为该猜想证明的里程碑。
- 2003年，Pintz a和Ruzsa对该猜想取得了最新的进展，但至今该猜想尚未得到完全证明。

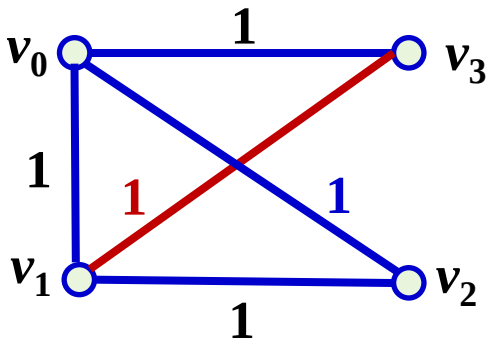


陈景润(1933.05-1996.03)

图的m着色问题(5)

- **图的m可着色问题的回溯法设计准备：**

1) 无向连通图G用邻接矩阵(Adjacency Matrix) a表示;



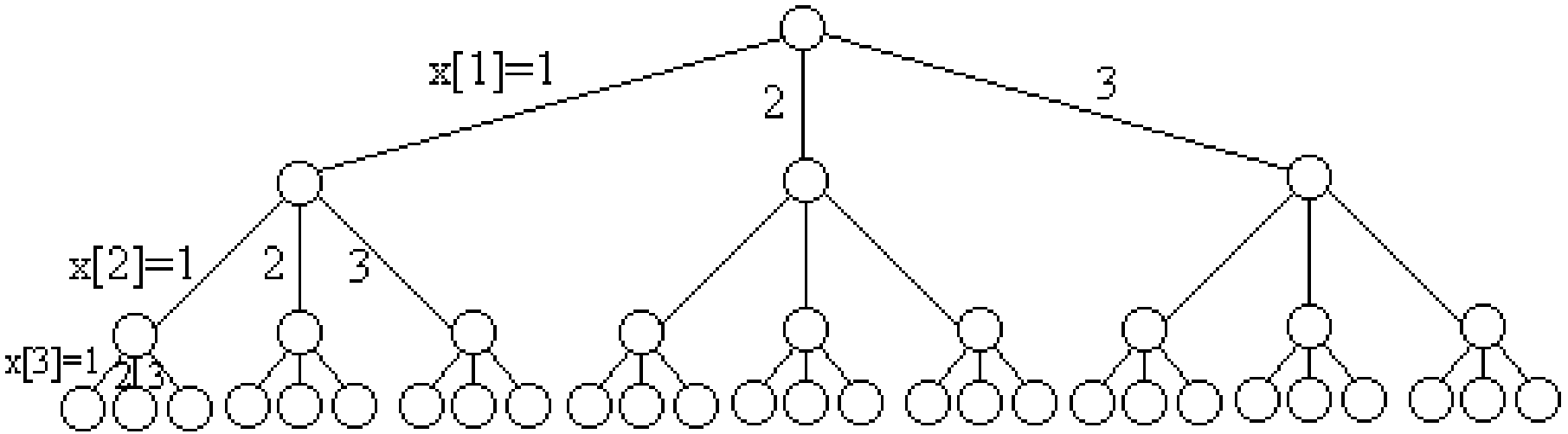
$$a=\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

2) 顶点i所着的颜色用x[i]表示，数组x[1:n]表示解向量。

3) 整数1,2,…,m用于表示m种不同颜色。

图的m着色问题(6)

- **图的m可着色问题的解空间树：**一颗高度为n+1的完全m叉树。解空间的第i层的每个结点都有m个儿子。每个儿子对应着一种着色。



n=3和m=3时的解空间树

图的m着色问题(7)

- 回溯法实现-主类、约束判断和主程序：

```
class Color  {
   friend int mColoring(int,int,int**);
private:
   bool OK(int); //约束函数
   void BackTrack(int); //核心函数
   int n, m;  //n顶点数，m可用颜色数
   int ** a;  //邻接矩阵
   int* x;  //当前解
   long sum;  //解数量
};
```

```
bool Color::OK(int k) {
   for(int j = 1; j <= n; j++)
      if( (a[k][j] == 1) &&
          (x[j] == x[k]) )  return false;
   return true;
}
```

```
int mColoring(int n,int m,int** a)  {
   Color z;
   z.n = n;
   z.m = m;
   z.a = a;
   z.sum = 0;

   int* p = new int[n+1];
   for(int i = 0; i <= n; i++)
      p[i] = 0;
   z.x = p;

   z.BackTrack(1);
   delete []p;
   return z.sum;
}
```

图的m着色问题(8)

- 回溯法实现-核心函数BackTrack：

```
void Color::BackTrack(int t) {
    if( t > n ) {
        sum ++;
    }else{
        for(int i = 1; i<= m; i++){
            x[t] = i;
            if( OK(t)) BackTrack(t+1);
            x[t] = 0;
        }
    }
}
```

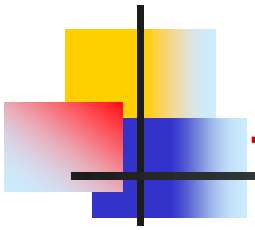
算法复杂度分析：

1)图m可着色问题的解空间树中内结点个数为：$\sum_{i=0}^{n-1} m^i$

2)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$O(mn)$。

3) 回溯法的算法复杂度为：

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



主要内容

- 回溯法的算法框架
- 装载问题
- n后问题
- 0-1背包问题
- 图的*m*着色问题
- 旅行售货员问题
- 连续邮资问题
- 回溯法效率分析

旅行售货员问题(1)

- 旅行售货员问题(Traveling Salesman Problem)：解空间是一颗排列树。
- 城市之间的连接用连通图G表示，连通图G用邻接矩阵a表示，a[i,j]的值为城市i和城市j之间的费用。

```
class Traveling{
  friend int TSP(int **,int [],int ,int);
  private:
  void Backtrack(int i);
  int n,     //图G的顶点数
      *x,      //当前解
      *bestx;  //当前最优解
  int **a,    //图G的邻接矩阵
      cc,     //当前费用
      bestc,   //当前最优值
      NoEdge;   //无边标记
};
```

```
int TSP(int **a,int v[],int n,int NoEdge){
  Traveling Y;   //定义Y
   Y.x=new int [n+1];
   for (int i=1;i<=n;i++)  Y.x[i]=i;
   Y.a=a; Y.n=n;
   Y.bestc=NoEdge;
   Y.bestx=v; Y.cc=0;
   Y.NoEdge=NoEdge;
   Y.Backtrack(2); //搜索x[2:n]的全排列
   delete[]Y.x;
   return Y.bestc;
}
```

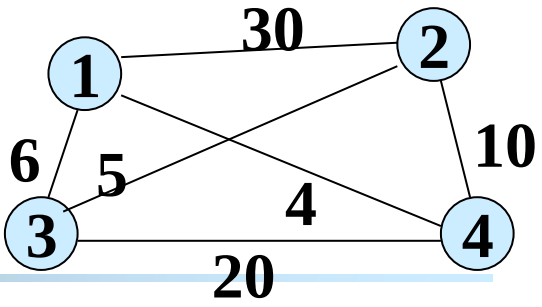
旅行售货员问题(2)

- 核心函数BackTrack的实现：

```
void Traveling::Backtrack(int i){
  if(i==n){ //排列结束
    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) {
       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++){   //可否进入x[j]子树，如果可以，则搜索子树
       if(a[x[i-1]][x[j]]!=NoEdge && (cc+a[x[i-1]][x[j]]<bestc || bestc==NoEdge)){
          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]);
       }
     }
  }
}
```

算法复杂度为：$O(n!)$

int n, //图G的顶点数 int **a, //图G的邻接矩阵
*x, //当前解 cc, //当前费用
*bestx; //当前最优解 bestc, //当前最优值
NoEdge; //无边标记



旅行售货员问题(3)

1) Y.a=a; Y.n=4; Y.bestc=NoEdge; Y.bestx=v; Y.cc=0; Y.NoEdge=NoEdge;

BackTrack(2) i=2 j=2 *a[x[2-1]][x[2]] != NoEdge*
&& (cc+a[x[2-1]][x[2]]<bestc || bestc==NoEdge)

swap(x[2],x[2]); cc+=a[x[2-1]][x[2]]=30 ⇩ 搜索子树 BackTrack(2+1)

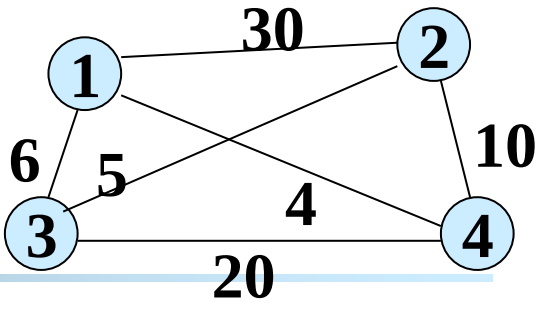
BackTrack(3) i=3 j=3 *a[x[3-1]][x[3]] != NoEdge*
&& (30+a[x[3-1]][x[3]]<bestc || bestc==NoEdge)

swap(x[3],x[3]); cc+=a[x[3-1]][x[3]]=35 ⇩ 搜索子树 BackTrack(3+1)

BackTrack(4) i=4 n=4 *a[x[n-1]][x[n]] != NoEdge*
&& (a[x[n]][1] != NoEdge)
&& (35+a[x[n-1]][x[n]]<bestc || bestc==NoEdge)

for(int j=1;j<=n;j++) bestx[j]=x[j]; bestx[ ]={1, 2, 3, 4}
bestc=35+a[x[n-1]][x[n]]+a[x[n]][1]; bestc= 35 + 20 + 4= 59

int n, //图G的顶点数 int **a, //图G的邻接矩阵
*x, //当前解 cc, //当前费用
*bestx; //当前最优解 bestc, //当前最优值
NoEdge; //无边标记



旅行售货员问题(4)

2) Y.a=a; Y.n=4; Y.bestc=59; Y.bestx [ ]={1, 2, 3 ,4}; Y.cc=35; Y.NoEdge=NoEdge;

BackTrack(3+1) i=3 j=3 *此时回溯*

cc -= a[x[3-1]][x[3]]=35-5=30; swap(x[3],x[3]); //{1, 2, 3 ,4}

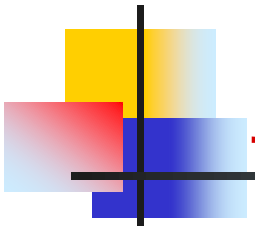
j++=4

a[x[3-1]][x[4]] != NoEdge
&& (30+a[x[3-1]][x[4]]<bestc || bestc==NoEdge)

swap(x[3],x[4]); // {*1, 2, 4, 3*} → BackTrack(3+1)
cc+=a[x[3-1]][*x[3]*]=30+10=40

BackTrack(4) i=4 n=4 *a[x[n-1]][x[n]] != NoEdge*
&& (a[x[n]][1] != NoEdge)
&& (40 +a[x[n-1]][x[n]] <bestc || bestc==NoEdge)

↓条件不满足，此处被限界处理掉!!!



主要内容

- 回溯法的算法框架
- 装载问题
- n后问题
- 0-1背包问题
- 图的*m*着色问题
- 旅行售货员问题
- 连续邮资问题
- 回溯法效率分析

连续邮资问题(1)

- 连续邮资问题描述：假设某国家发行了*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.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,使得可在1张信封上贴出从邮资1开始,增量为1的最大连续邮资区间。

例如:当*n*=5, *m*=4时, 最佳设计为 (1,3,11,15,32)的五种邮票面值，此时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(2)

- 算法设计：

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序，最多可帖m张邮票，具体排列如下：

1）初始：$x[1]=1$是唯一的选择； 此时最大邮资区间为$[1:m]$

2）显然， $x[2]$的一个合适取值区间为$[2: m+1]$；（！！关键点1）

3）以此类推，一般情况有：

假设已选定$x[1,i-1]$的值，其对应的最大连续邮资区间为$[1:r]$，则$x[i]$可能取值区间为：$[x[i-1]+1，r+1]$。

由此可见，该问题的解空间确定，可用回溯法搜索。

连续邮资问题(3)

- 算法实现：主类设计

```
class Stamp {
    friend int MaxStamp(int  ,int  ,int []);
     void Backtrack(int i,int r);
    int n; //邮票面值数
     int m; //每张信封最多允许贴的邮票数
     int maxvalue; //当前最优值
     int maxint; //大整数
     int maxl; //邮资上界
     int *x; //当前解
     int *y; //贴出各种邮资所需最少邮票
     int *bestx; //当前最优解
}
```

连续邮资问题(4)

- **深度搜索策略解析：**

在函数Backtrack中，当***i*>*n***时，表示算法已搜索至一个叶结点，得到一个新的邮票面值设计方案***x*[1:*n*]**。如果该方案能贴出的最大连续邮资区间大于当前已找到的最大连续邮资区间**maxvalue**，则更新当前最优值**maxvalue**和相应的最优解**bestx**。

当***i*<=*n***时，当前扩展结点***Z***是解空间中的一个内部结点。**在该结点处*x*[1:*i*-1]能贴出的最大连续邮资区间为*r*-1。因此，在结点*Z*处，*x*[*i*]的可取值范围是[*x*[*i*-1]+1:*r*]，从而，结点*Z*有*r*-*x*[*i*-1]个儿子结点。算法对当前扩展结点*Z*的每个儿子结点，以深度优先的方式递归的对相应的子树进行搜索。**

由于***x*[*i*]**的取值范围已经确定**[*x*[*i*-1]+1, *r*]**，则对于一个确定的***x*[*i*]**，关键问题就变成了：**如何更新*r*的值，即*x*[1…*i*]能表示的最大连续邮资区间。**

连续邮资问题(5)

- 算法实现：主函数MaxStamp

```
int MaxStamp (int n,int m,int bestx[]) {
      Stamp X;
      int maxint=32767;  int maxl=1500;
      X.n=n;  X.m=m;  X.maxvalue=0;  X.maxint=maxint;
      X.maxl=maxl;  X.bestx=bestx;
      X.x=new int [n+1];  X.y=new int [maxl+1];
      for(int i=0;i<=n;i++)   X.x[i]=0;
      for(i=1;i<=maxl;i++)   X.y[i]=maxint;
      X.x[1]=1;  X.y[0]=0;
      X.Backtrack(2,1);
      delete[] X.x;  delete [] X.y;
      return X.maxvalue;
}
```

连续邮资问题(6)

- **关于更新r的值：**

考虑x[i]加入后对当前状态的影响：

如果原有情况中贴的邮票不满m张，那就一直贴x[i]，直到达到m张邮票，那么在这个过程会产生出很多不同的邮资，这些邮资都应该被加入到新情况中。

引入数组y定义：记录当前已选定的邮票面值x[1:i]能贴出各种邮资所需的最少邮票张数。即：y[k]表示是用不超过m张面值为x[1:i]的邮票贴出邮资k所需的最少邮票数。（！！关键点2）

因为原有情况已经对y[k]进行了计算，现在的工作就变成了：根据新产生的邮资所需要的最少邮票张数对y[ ]数组进行更新。

（！！关键点2）数组y：记录当前已选定的邮票面值x[1:i]能贴出各种邮资所需的最少邮票张数。即：*y*[*k*]表示是用不超过*m*张面值为*x*[1:*i*]的邮票贴出邮资*k*所需的最少邮票数。

连续邮资问题(7)

- 算法实现：核心函数BackTrack(int i, int r)

```
void Stamp::Backtrack(int i, int r) {
 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++; //找到最大r
 if (i>n){ //到达叶节点
     if(r-1>maxvalue){
       maxvalue=r-1;
       for(int j=1;j<=n;j++)
         bestx[j]=x[j];
     }
     return;
 }
```

```
  int *z=new int[maxl+1];

  for(int k=1; k<=maxl; k++)
    z[k]=y[k];   //保存现场

  for(j=x[i-1]+1; j<=r; j++){
    x[i]=j;//在解空间内逐个尝试
    Backtrack(i+1,r);

    for(int k=1; k<=maxl; k++)
      y[k]=z[k];  //恢复现场
  }
  delete[] z;
}
```

$n=5, m=4$ n种不同面值的邮票，每张信封上最多只允许贴m张.

$y[k]$表示是用不超过m张面值为$x[1:i]$的邮票贴出邮资k所需的最少邮票数

```
if(y[j]+k<y[j+x[i-1]*k])
    y[j+x[i-1]*k]=y[j]+k;
```

连续邮资问题(8)

1. BackTrack(2, 1) $i=2$, $r=1$, maxValue=0 $x[1]=1$, $y[0]=0$

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]
0	1	0	0	0	0

y[0]	y[1]	y[2]	y[3]	y[4]	…
0	∞	∞	∞	∞	∞
0	1	∞	∞	∞	∞
0	1	2	∞	∞	∞
0	1	2	3	∞	∞
0	1	2	3	4	∞

$j \in [0, x[i-2]*(m-1)]$，即： $j \in [0, 0]$

$j=0$ $y[j]=0$ → $k \in [1, 4-0]$即[1, m-y[j]]

$k=1$ $y[0]+1<y[0+x[1]*1]=\infty$

$y[0+x[1]*1]=y[1]=y[0]+1=1$

$k=2$ $y[0]+2<y[0+x[1]*2]=\infty$

$y[0+x[1]*2]=y[2]=y[0]+2=2$

$k=3$ $y[0]+3<y[0+x[1]*3]=\infty$

$y[0+x[1]*3]=y[3]=y[0]+3=3$

$k=4$ $y[0]+4<y[0+x[1]*4]=\infty$

$y[0+x[1]*4]=y[4]=y[0]+4=4$

由此可得 $r=4+1$，即 $x[2] \in [x[1]+1, 4+1]$

$n=5, m=4$ n种不同面值的邮票，每张信封上最多只允许贴m张.

$y[k]$表示是用不超过m张面值为$x[1:i]$的邮票贴出邮资k所需的最少邮票数

连续邮资问题(9)

```
if(y[j]+k<y[j+x[i-1]*k])
    y[j+x[i-1]*k]=y[j]+k;
```

2.1 BackTrack(2+1, 5) $i=2+1=3$, $r=4+1=5$, maxValue=0。开始尝试解空间：$x[2]=2$

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]
0	1	2	0	0	0

y[0]	y[1]	y[2]	y[3]	y[4]	y[5]	y[6]	y[7]	y[8]
0	1	2	3	4	∞	∞	∞	∞
0	1	1	3	4	∞	∞	∞	∞
0	1	1	3	2	∞	∞	∞	∞
0	1	1	3	2	∞	3	∞	∞
0	1	1	3	2	∞	3	∞	4

$j \in [0, x[i-2]*(m-1)]$，即： $j \in [0, 3]$

$j=0$ $y[j]=0$ $\rightarrow$ $k \in [1, 4-0]$

k=1 $y[0]+1 < y[0+x[2]*1] = 2$

$y[0+x[2]*1]=y[2]=y[0]+1=1$

k=2 $y[0]+2 < y[0+x[2]*2] = 4$

$y[0+x[2]*2]=y[4]=y[0]+2=2$

k=3 $y[0]+3 < y[0+x[2]*3] = \infty$

$y[0+x[2]*3]=y[6]=y[0]+3=3$

k=4 $y[0]+4 < y[0+x[2]*4] = \infty$

$y[0+x[2]*4]=y[8]=y[0]+4=4$

$n=5, m=4$ n种不同面值的邮票，每张信封上最多只允许贴m张.

$y[k]$表示是用不超过m张面值为$x[1:i]$的邮票贴出邮资k所需的最少邮票数

连续邮资问题(10)

```
if(y[j]+k<y[j+x[i-1]*k])
    y[j+x[i-1]*k]=y[j]+k;
```

2.2 BackTrack(2+1, 5) $i=2+1=3$, $r=4+1=5$, maxValue=0。开始尝试解空间：$x[2]=2$

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]
0	1	2	0	0	0

y[0]	y[1]	y[2]	y[3]	y[4]	y[5]	y[6]	y[7]	y[8]
0	1	2	3	4	∞	3	∞	4
0	1	1	2	4	∞	3	∞	4
0	1	1	2	2	3	3	∞	4
0	1	1	2	2	3	3	4	4

$j \in [0, x[i-2]*(m-1)]$，即： $j \in [0, 3]$

$j=1$ $y[1]=1 \rightarrow k \in [1, 4-1]$

$k=1$ $y[1]+1 < y[1+x[2]*1]= 3$

$y[1+x[2]*1]=y[3]=y[1]+1=2$

$k=2$ $y[1]+2 < y[1+x[2]*2]= \infty$

$y[1+x[2]*1]=y[5]=y[1]+2=3$

$k=3$ $y[1]+3 < y[1+x[2]*3]= \infty$

$y[1+x[2]*3]=y[7]=y[1]+3=4$

$n=5, m=4$ n种不同面值的邮票，每张信封上最多只允许贴m张.

$y[k]$表示是用不超过m张面值为$x[1:i]$的邮票贴出邮资k所需的最少邮票数

连续邮资问题(11)

```
if(y[j]+k<y[j+x[i-1]*k])
    y[j+x[i-1]*k]=y[j]+k;
```

2.3 BackTrack(2+1, 5) $i=2+1=3$, $r=4+1=5$, maxValue=0。开始尝试解空间：$x[2]=2$

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]
0	1	2	0	0	0

y[0]	y[1]	y[2]	y[3]	y[4]	y[5]	y[6]	y[7]	y[8]
0	1	1	2	2	3	3	4	4

$j\in[0, x[i-2]*(m-1)]$，即： $j\in[0, 3]$

$j=2$ $y[2]=1$ $\rightarrow$ $k\in[1, 4-1]$

$k=1$ $y[2]+1 = = y[2+x[2]*1]= 2$

张数一样，不用更新！

$k=2$ $y[2]+2 = = y[2+x[2]*2]= 3$

张数一样，不用更新！

$k=3$ $y[2]+3 = = y[2+x[2]*3]= 4$

张数一样，不用更新！

n=5, *m*=4　*n*种不同面值的邮票，每张信封上最多只允许贴*m*张.

y[*k*]表示是用不超过*m*张面值为*x*[1:*i*]的邮票贴出邮资*k*所需的最少邮票数

连续邮资问题(12)

2.4　BackTrack(2+1, 5)　*i*=2+1=3, *r*=4+1=5, maxValue=0。开始尝试解空间：*x*[2]=2

x[0]	*x*[1]	*x*[2]	*x*[3]	*x*[4]	*x*[5]
0	1	2	0	0	0

y[0]	*y*[1]	*y*[2]	*y*[3]	*y*[4]	*y*[5]	*y*[6]	*y*[7]	*y*[8]	*y*[9]
0	1	1	2	2	3	3	4	4	∞

j∈[0, *x*[i-2]*(*m*-1)]，即：　j∈[0, 3]

j=3　*y*[3]=2　⟶　k∈[1, 4-2]

k=1　*y*[3]+1 = = *y*[3+*x*[2]*1]= 3

张数一样，不用更新！

k=2　*y*[3]+2 = = *y*[3+*x*[2]*2]= 4

张数一样，不用更新！

while(y[r]<maxint) r++;

由此可得 *r*=8+1=9，即 *x*[3] ∈ [*x*[2]+1, 8+1]

$n=5, m=4$ n种不同面值的邮票，每张信封上最多只允许贴m张.

$y[k]$表示是用不超过m张面值为$x[1:i]$的邮票贴出邮资k所需的最少邮票数

连续邮资问题(13)

3.1 BackTrack(3+1, 9) $i=3+1=4$, $r=8+1=9$, maxValue=0。开始尝试解空间：$x[3]=3$

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]
0	1	2	3	0	0

y[0]	y[1]	y[2]	y[3]	y[4]	y[5]	y[6]	y[7]	y[8]	y[9]
0	1	1	2	2	3	3	4	4	∞
0	1	1	1	2	3	3	4	4	∞
0	1	1	1	2	3	2	4	4	∞
0	1	1	1	2	3	2	4	4	3

j∈[0, $x[i-2]*(m-1)$]，即： j∈[0, 6]

j=0 $y[j]=0$ ⟶ k∈[1, 4-0]

k=1 $y[0]+1< y[0+x[3]*1]= 2$

$y[0+x[3]*1]=y[3]=y[0]+1=1$

k=2 $y[0]+2< y[0+x[3]*2]= 3$

$y[0+x[3]*2]=y[6]=y[0]+2=2$

k=3 $y[0]+3< y[0+x[3]*3]= \infty$

$y[0+x[3]*3]=y[9]=y[0]+3=3$

k=4 $y[0]+4< y[0+x[3]*4]= \infty$

$y[0+x[3]*4]=y[12]=y[0]+4=4$

y[0]	y[1]	y[2]	…	y[9]	y[10]	y[11]	y[12]	y[13]	y[14]
0	1	1	…	3	∞	∞	∞	∞	∞
0	1	1	…	3	∞	∞	4	∞	∞

$n=5, m=4$ n种不同面值的邮票，每张信封上最多只允许贴m张.

$y[k]$表示是用不超过m张面值为$x[1:i]$的邮票贴出邮资k所需的最少邮票数

连续邮资问题(14)

3.2 BackTrack(3+1, 9) $i=3+1=4$, $r=8+1=9$, maxValue=0。开始尝试解空间：$x[3]=3$

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]
0	1	2	3	0	0

y[0]	y[1]	y[2]	y[3]	y[4]	y[5]	y[6]	y[7]	y[8]	y[9]
0	1	1	1	2	3	2	4	4	3

3

y[10]	y[11]	y[12]	y[13]	y[14]	y[15]	y[16]	y[17]	y[18]	y[19]
∞	∞	4	∞	∞	∞	∞	∞	∞	∞

4

j∈[0, x[i-2]*(m-1)]，即： j∈[0, 6]

j=1 y[j]=1 → k∈[1, 4-1]

k=1 y[1]+1 = = y[1+x[3]*1]= 2

张数一样，不用更新！

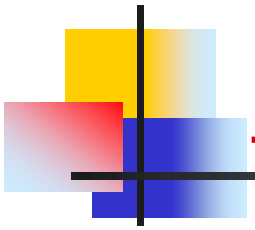
k=2 y[1]+2 < y[1+x[3]*2]= 4

y[1+x[3]*2]=y[7]=3

k=3 y[1]+3 < y[1+x[3]*3]=∞

y[1+x[3]*3]=y[10]=4

To Be Continued…



主要内容

- 回溯法的算法框架
- 装载问题
- n后问题
- 0-1背包问题
- 图的*m*着色问题
- 旅行售货员问题
- 连续邮资问题
- 回溯法效率分析

回溯法的效率分析(1)

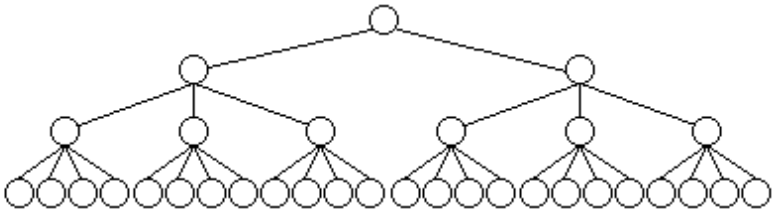
回溯算法的效率在很大程度上依赖于以下因素：

(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。

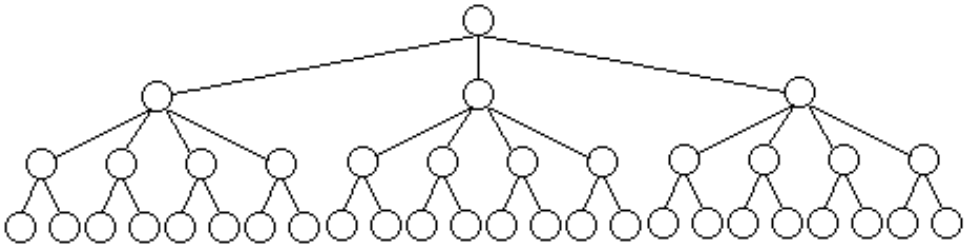
好的约束函数能显著地减少所生成的结点数。但这样的约束函数往往计算量较大。因此，在选择约束函数时通常存在生成结点数与约束函数计算量之间的折衷。

回溯法的效率分析(2)

- **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。



(a)

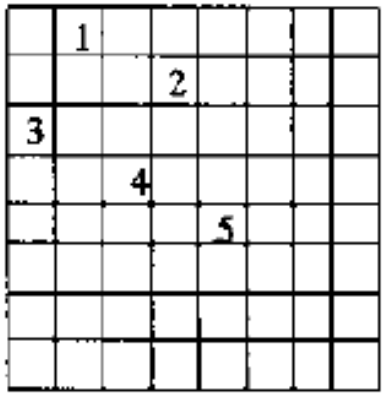


(b)

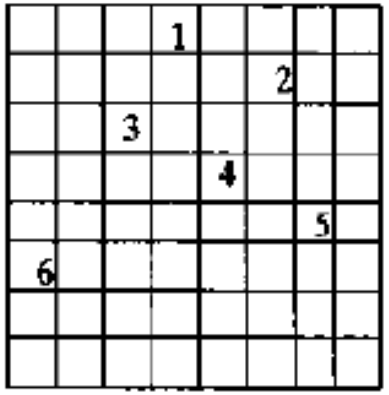
图(a)中，从第1层剪去1棵子树，则从所有应当考虑的3元组中一次消去12个3元组。对于图(b)，虽然同样从第1层剪去1棵子树，却只从应当考虑的3元组中消去8个3元组。前者的效果明显比后者好。

- **概率法**：使用概率法估计回溯法在解具体实例时所产生的结点个数。主要思想是通过在解空间树上产生一条随机路径，然后沿此路径估算解空间树中满足约束条件的结点总数（8皇后问题）。

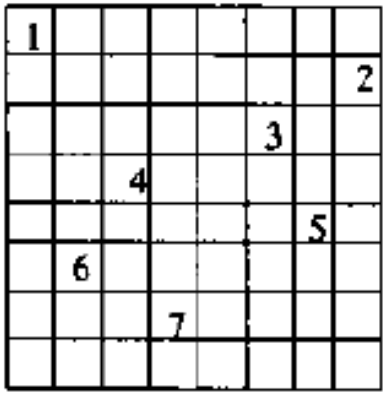
回溯法的效率分析(3)



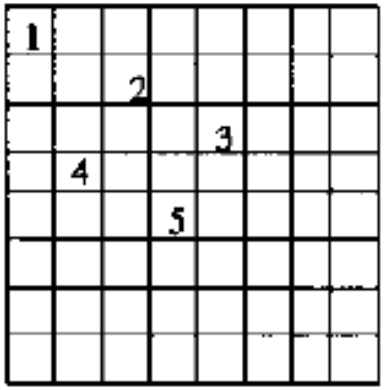
(8,5,4,3,2)=1649



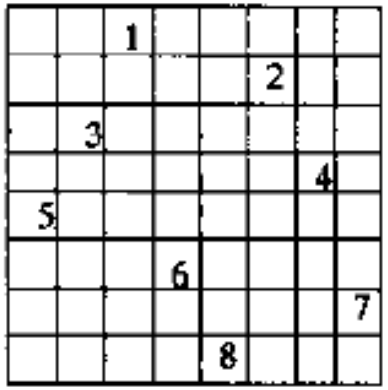
(8,5,3,1,2,1)=769



(8,6,4,2,1,1,1)=1785

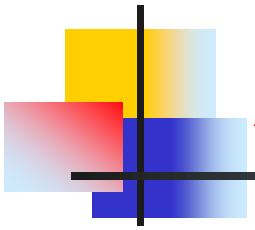


(8,6,4,3,2)=1977



(8,5,3,2,2,1,1,1)=2329

图 5-11　解空间树中 5 条随机路径所对应的棋盘状态



小结

- 回溯法的算法框架：解空间，回溯，剪枝
- 装载问题：子集树
- *n*后问题：子集树
- 0-1背包问题：子集树
- 图的*m*着色问题：子集树
- 旅行售货员问题：排列树
- 连续邮资问题： 子集树
- 回溯法效率分析

谢谢大家！