4.回溯

深度优先搜索策略，根据约束函数和限界函数剪枝

$$\textbf{解空间结构}\begin{cases}\text{子集树}O(2^n):\text{选}/\text{不选，}0-1\text{背包}\\ \text{排列树}O((n!):\text{顺序相关，旅行商问题}\end{cases}$$

- 回溯法伪代码：

```
void BackTrack(int t){
    if(t>n) Output(x);
    else
        for(int i=f(n,t),int j=g(n,t);i++){
        //当前扩展结点处未搜索过的子树起始与终止编号
            x[t]=h(i);//当前扩展节点处x[t]的第i个可选值
            if (COnstraint(t) && Bound(t)) BackTrack(t+1);
        }
}
```

- 子集树
- ```
  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);
        }
    }
  }
  ```
- 排列树
- ```
  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]);
        }
    }
  }
  ```

剪枝函数

$\begin{cases}\text{约束函数：减去不满足约束的子树} \\ \text{限界函数：减去不能得到最优解的子树（先能得到解再考虑后面的} \end{cases}$

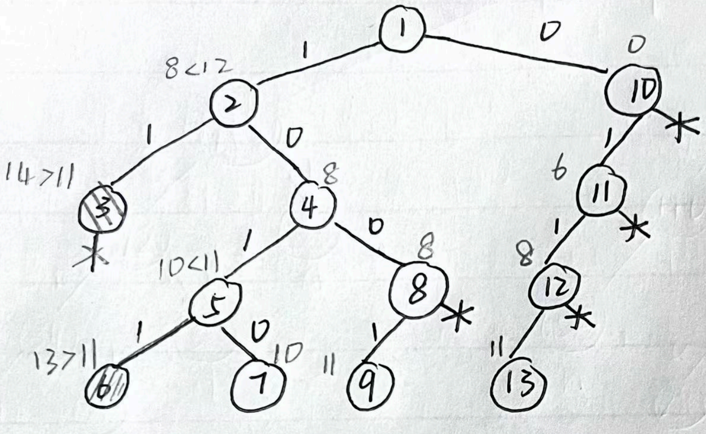
装载（子集树）

装载问题

最优装载方案：将第1艘轮船尽可能装满，将剩余货箱装到第2艘轮船

约束函数：$\sum_{i=1}^{n} w_i x_i < C_1$

例：$n=4$，$C_1=12$，$W=\{8,6,2,3\}$



约束剪左枝

限界剪右枝

(1) 定义一个Loading类

私有变量（w,c,n,bestw,cw）

和函数 Backtrack()

(2)

```
Type Maxloading(type w[],
type c, int n,)
    Loading <Type> X;
    //初始化X
    X. w=w;  //集装箱重量数组
    X. c=c;   //第一艘船载重量
    X. n=n;  //集装箱数
    X. bestw=0; //最优载重
    X. cw=0; //当前载重量
    X.Backtrack(1);  //搜索树
    return X.bestw;   }
```

(3)

```
void Loading<Type>::Backtrack(int i)
{ / /搜索第i层结点
   if (i>n) {//到达叶结点
          if (cw>bestw)  bestw=cw;
        return;  }
   //搜索子树
   if (cw+w[i]<=c){ //x[i]=1
        cw += w[i];
        Backtrack (i+1);
        cw - = w[i]; }

     Backtrack(i+1);  //x[i]=0
}
```

上界函数-解决右子树剪枝问题

```
template < class Type >
Type Maxloading(type w[],
type c, int n,)
    loading <Type> X;
   //初始化X
   X. w=w;  //集装箱重量数组
   X. c=c;   //第一艘船载重量
   X. n=n;  //集装箱数
   X. bestw=0; //当前最优载重
   X. cw=0; //当前载重量
    X.r=0; //剩余集装箱重量
   for (int i=1; i<=n; i++)
         X. r +=w[i]
 //计算最优载重量
   X.Backtrack(1);
   return X.bestw;   }
```

```
template<classType>
void Loading<Type>::Backtrack(int i)
{ / /搜索第i层结点
   if (i>n) {//到达叶结点
       bestw=cw;
       return; }
   //搜索子树
   r -= w[i];
   if (cw+w[i]<=c){ //x[i]=1
       cw += w[i];
       Backtrack (i+1);
       cw - = w[i]; }
   if (cw+r > bestw){ //x[i]=0
       Backtrack(i+1);
   r += w[i];
   }
```

- **算法复杂性**: $O(2^n)$

构造最优解-装载问题的回溯算法

```
template < class Type >
Type Maxloading(type w[],
type c, int n, int bestx[ ])
    loading <Type> X;
   //初始化X
    X.x=new int[n+1];
    X.bestx=0;
   X. w=w;  //集装箱重量数组
   X. c=c;   //第一艘船载重量
   X. n=n;  //集装箱数
   X. bestw=0; //当前最优载重
   X. cw=0; //当前载重量
   X. r=0; //剩余集装箱重量
   for (int i=1; i<=n; i++)
      X. r +=w[i]
 //计算最优载重量
   X.Backtrack(1);
   delete [ ] X.x     //释放X
   return X.bestw;   }
```

```
template<classType>
void Loading<Type>::Backtrack(int i)
{ / /搜索第i层结点
   if (i>n) {//到达叶结点
          {for(j=1;j<=n;j++)
                   bestx[j]=x[j];
       bestw=cw;  }
     return;  }
   //搜索子树
   r - = w[i];
   if (cw+w[i]<=c){
       x[i]=1
     cw += w[i];
     Backtrack (i+1);
     cw - = w[i]; }
    if (cw+r > bestw){
       x[i]=0
     Backtrack(i+1);
   r+=w[i]
   }
```

批作业处理（排列树）

批处理作业调度 — 排列树

F_{ji} = 作业i在机器j上完成处理的时间

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

顺序：1→3→2

$F_{11}=2$　　$F_{21}=3$

$F_{13}=4$ ---> $F_{23}=7$

$F_{12}=7$　　$F_{22}=8$

0-1背包（子集树）

0-1 背包

用贪心算法可以得到上界进行限界（得到可行解再考虑限界）

设：cw=背包当前重量　cp=当前装入背包的总价值

rp=用贪心计算剩余的最大收益　bp=cp+rp：上界

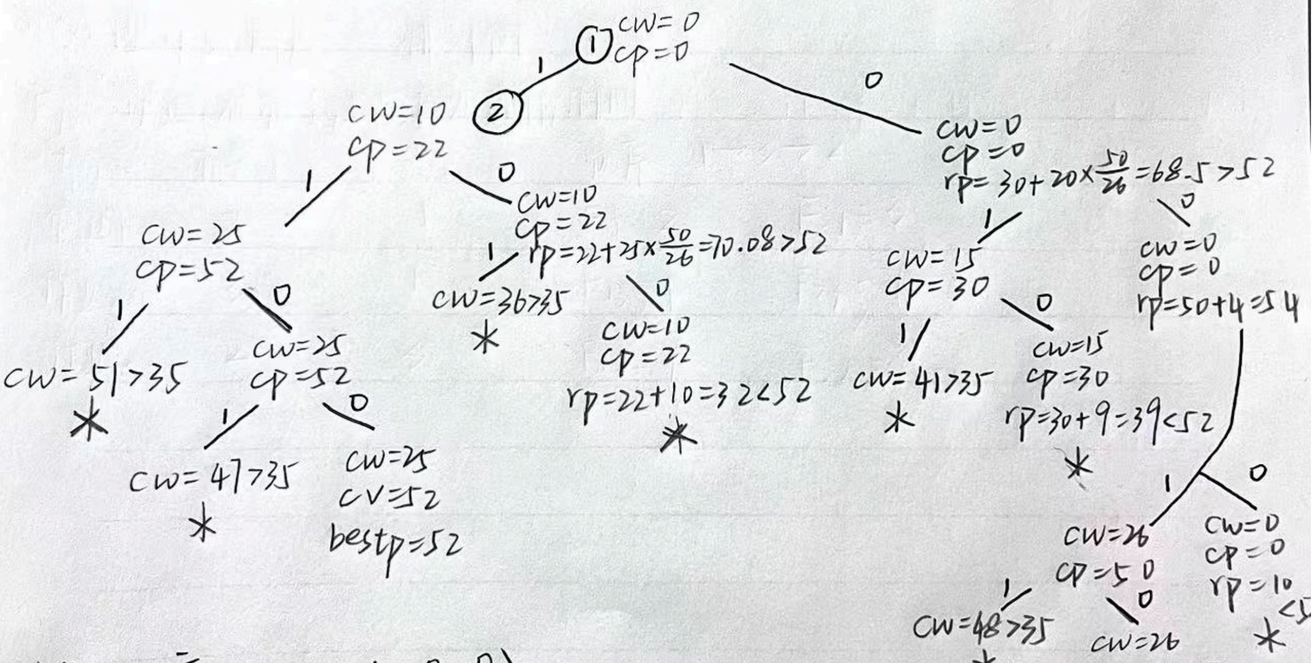
bestp=当前最优价值

★按价值排序后，最终回答序列记得恢复顺序

例：c=35，w={10,26,15,22}，v={22,50,30,10}

前置：按 $\frac{v}{w}$ 非增序排序：

	0	1	2	3
v	22	30	50	10
w	10	15	26	22
$\frac{v}{w}$	2.2	2	1.92	0.45



最后放入序列：(1,1,0,0)

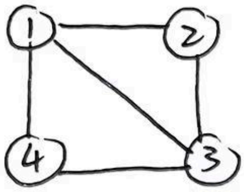
按原问题排序后：(1,0,1,0)

最大团（子集树）

最大团问题：节点最多的团

团：最大完全子图（完全图：图中每两点可达）

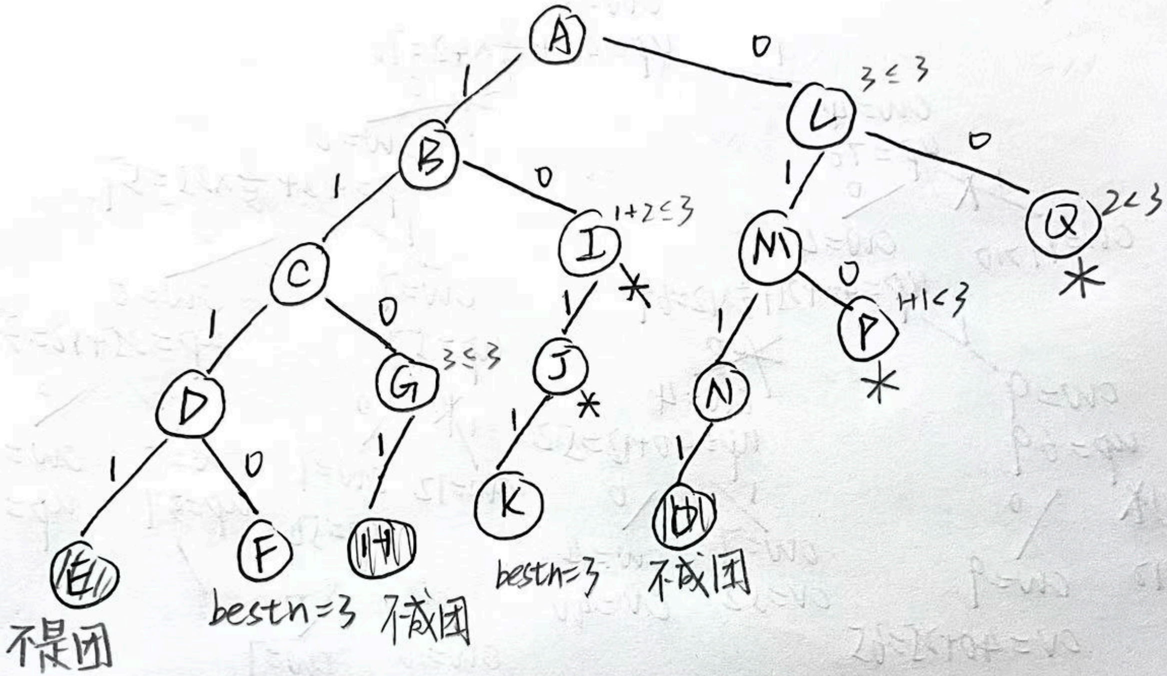
例：



约束条件：$\{x_1, x_2, \dots, x_i\} \cup \{x_{i+1}\}$ → 不是团
剪枝

限界函数：$cn + r \le bestn$ → 剪枝

cn：当前团尺寸　r：剩余节点数　bestn：已求出最大团尺寸



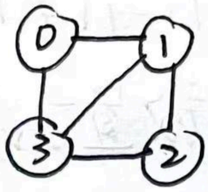
∴最大团尺寸为3：(1, 2, 3)，(1, 3, 4)

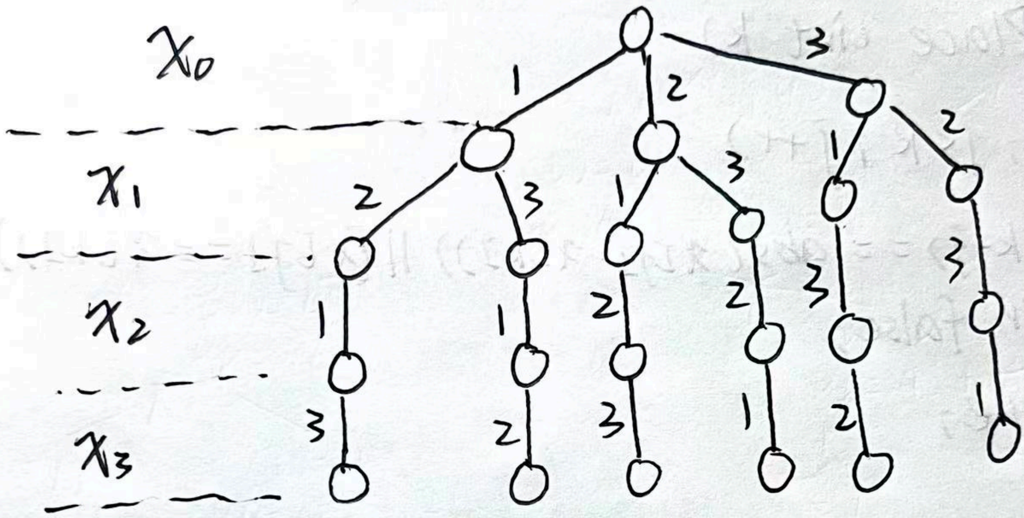
图的m着色

图的m着色（任意相邻两个结点有不同颜色）

四色定理 ⇒ m=3的可能性，找第一个解

例：给出3着色方案





图G所有可能的3-着色方案

着色问题回溯算法

```
int mColoring(int n, int m,  int **a  )
 { Color X;
  //初始化X
  X.  n=n;   //图的顶点数
  X.  m=m   //可用颜色数
  X.  a=a;   //图的邻接矩阵
  X.  Sum=0; //已找到的着色方案数
  int*p=new int [n+1];
  for (int i=0;  i<=n;  i++)
    p[i]=0;
  X. x=p    //当前解;
  X. Backtrack(1);
  delete [ ]p;
  returnX.  sum;  }
```

```
voidColor 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;
   retrun true;
```

算法复杂性：

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

n皇后（排列树）

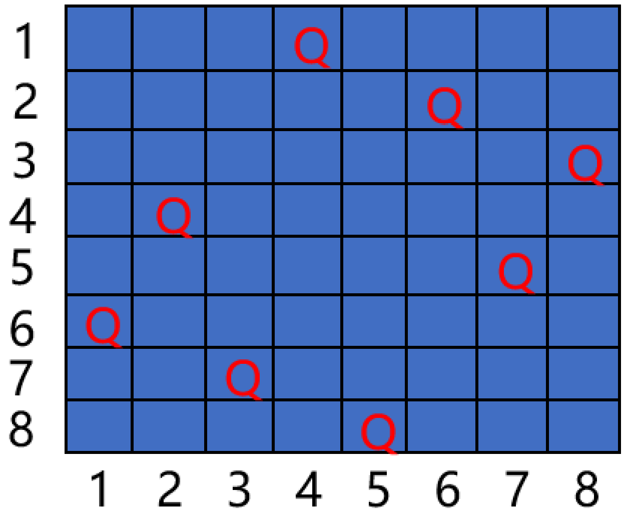
皇后问题要求在一个$n \times n$的棋盘上放置n个皇后，使得它们彼此不受“攻击”。n-皇后问题要求寻找在棋盘上放置这n个皇后的方案，使得它们中任何两个都不在同一行、同一列或同一斜线上。

输入：

给定棋盘的大小n (n ≤ 13)

输出：

输出有多少种放置方法。



算法思路:将棋盘从左至右,从上到下编号为1,...,n,皇后编号为1,...,n.

设解为(x_1, ..., x_n)，x_i为皇后i的列号,且x_i位于第i行.

解空间:E={ (x_1,..., x_n) | $x_i \in$Si, i=1,...,n}, Si={1, ..., n},1 ≤ i ≤ n

解空间为排列树.

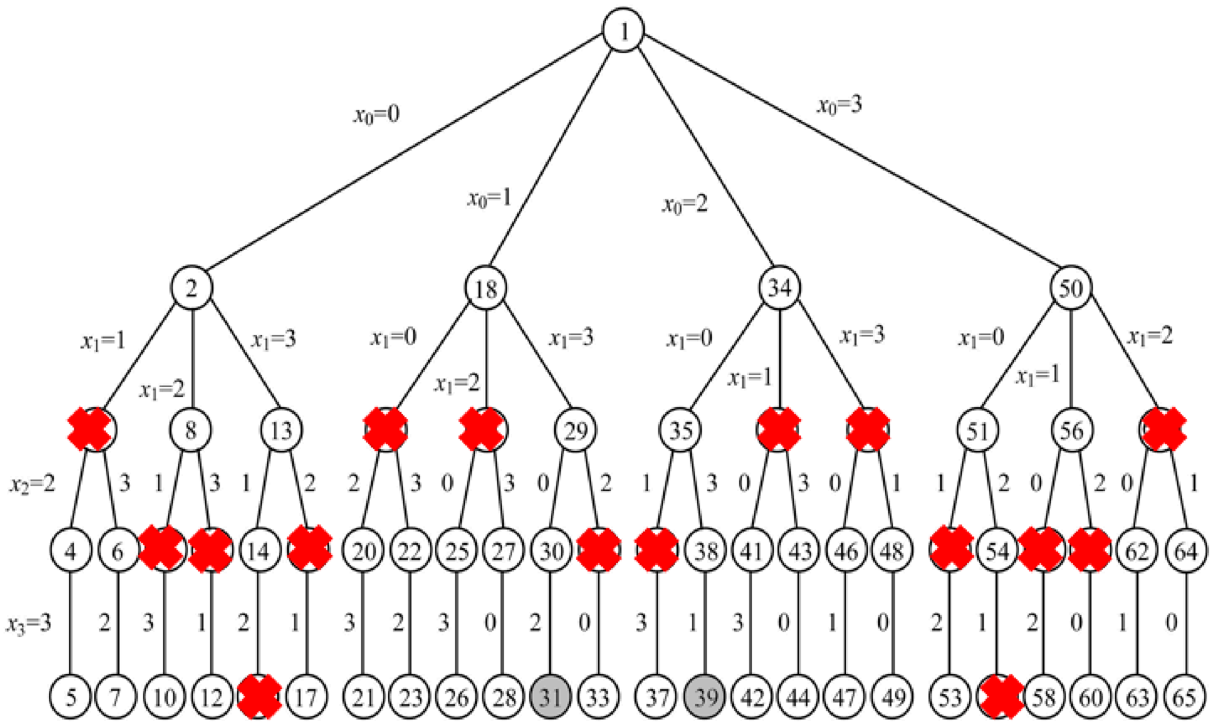
其约束集合D为

1) $x_i \neq x_j$　　皇后i, j不在同一列上

2) $x_i - i \neq x_j - j$

3) $x_i + i \neq x_j + j$

皇后i, j不在同一斜线上 abs(i-j) ≠ abs(x[i] – x[j]))



n后问题的回溯算法

```
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; }

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

```
int nQueen(int n)
   {
   QueenX;
   //初始化X
   X. n=n;  //皇后个数
   X. sum=0;
   int*p=new int [n+1];
   for(int i=0;  i<=n;  i++)
     p[i]= 0;
   X.x=p;
   X.Backtrack(1);
   delete [] p;
   returnX. sum;  }
```