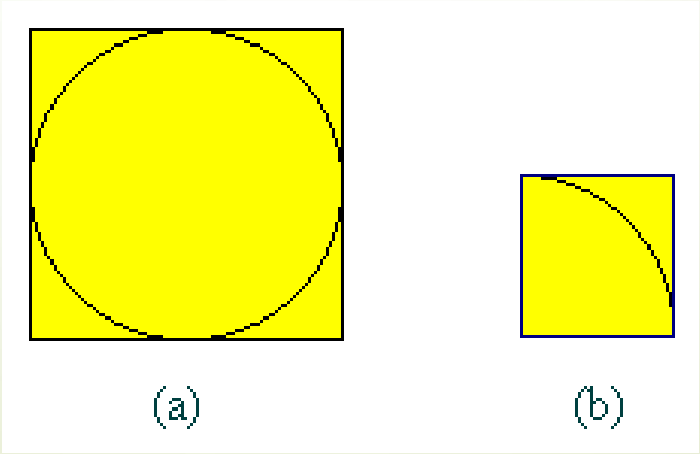
随机算法回顾

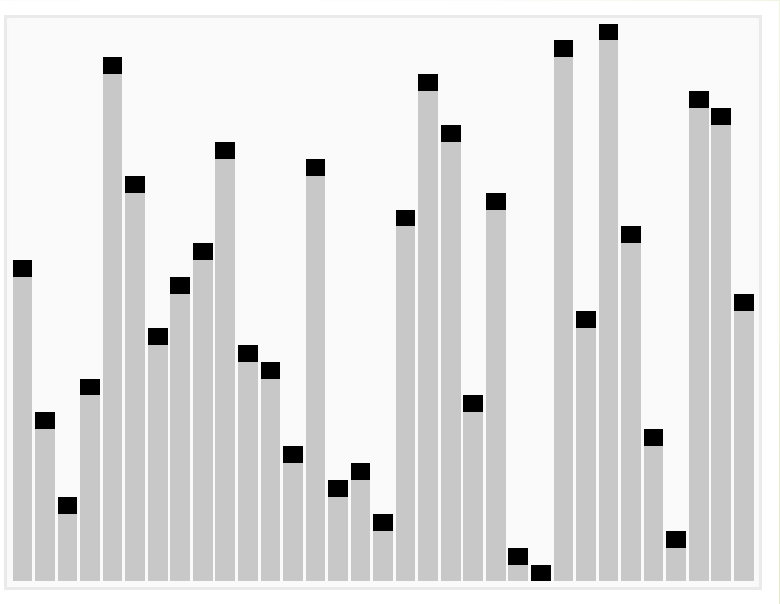
随机算法的分类

- 数值概率算法常用于数值问题的求解。将一个问题的计算与某个概率分布已经确定的事件联系起来，求问题的近似解。这类算法所得到的往往是近似解，且近似解的精度随计算时间的增加而不断提高。在许多情况下，要计算出问题的精确解是不可能或没有必要的，因此可以用数值随机化算法得到相当满意的解。



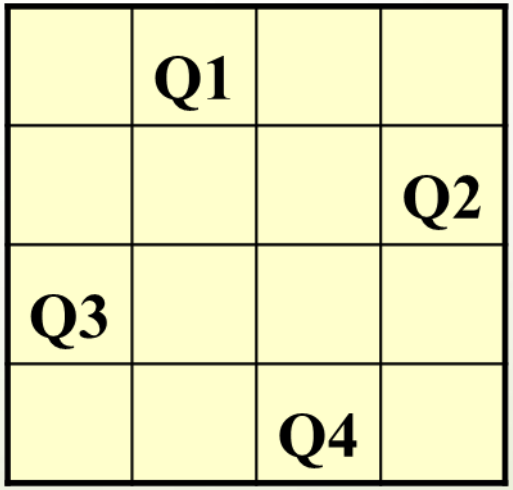
随机算法的分类

- 舍伍德算法总能求得问题的一个解，且所求得的解总是正确的。当一个确定性算法在最坏情况下的计算复杂性与其在平均情况下的计算复杂性有较大差别时，可在这个确定性算法中引入随机性将它改造成一个舍伍德算法，消除或减少问题的好坏实例间的这种差别（精髓所在）。



随机算法的分类

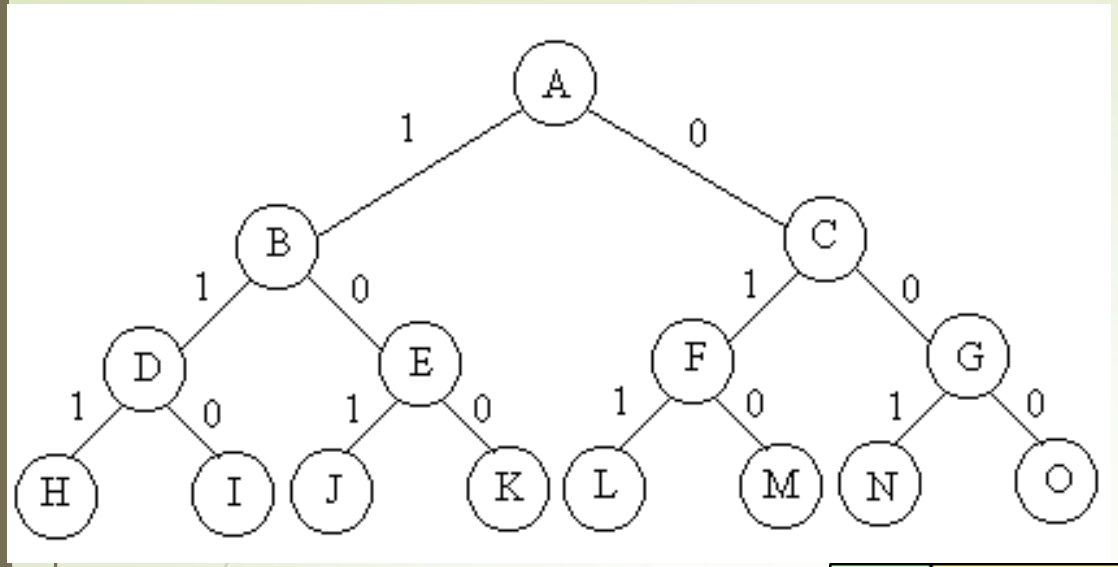
- 拉斯维加斯算法不会得到不正确的解。一旦用拉斯维加斯算法找到一个解，这个解就一定是正确解。但有时可能找不到解。拉斯维加斯算法找到正确解的概率随着它所用的计算时间的增加而提高。对于所求解问题的任意实例，用同一拉斯维加斯算法反复对它求解，可以使求解失效的概率任意小。

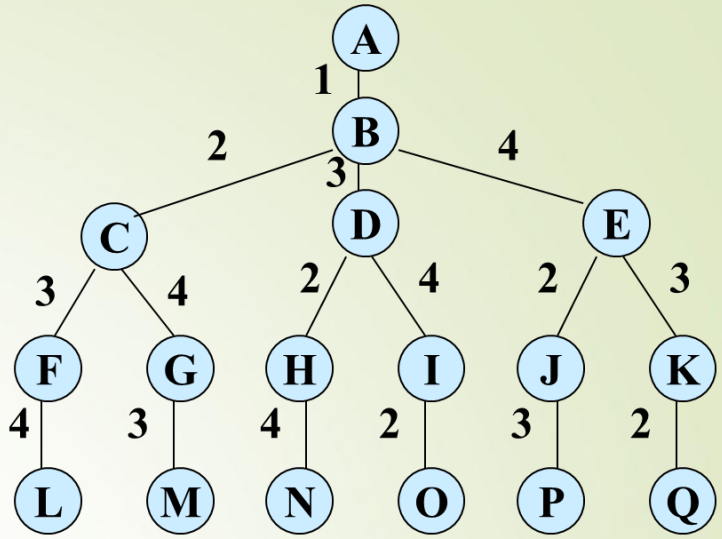


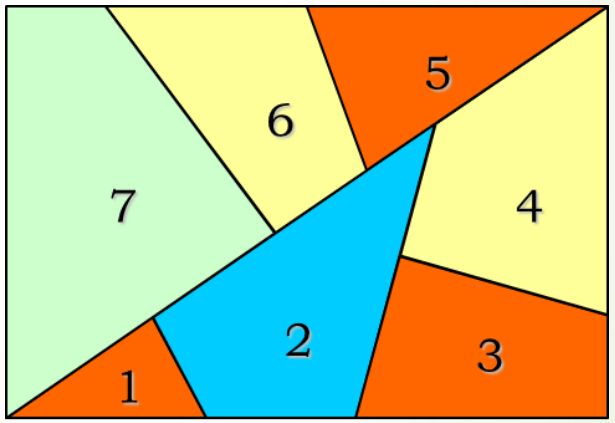
随机算法的分类

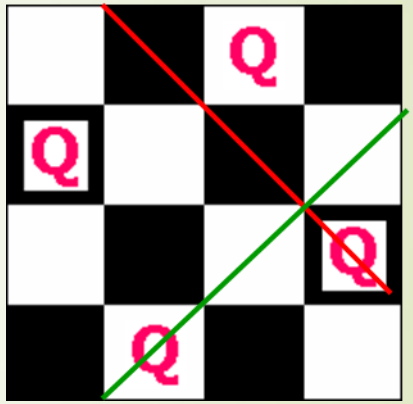
- 蒙特卡罗算法用于求问题的准确解，但得到的解未必是正确的。蒙特卡罗算法以正的概率给出正解，求得正确解的概率依赖于算法所用的时间。算法所用的时间越多，得到正确解的概率就越高。一般给定执行步骤的上界，给定一个输入，算法都是在一个固定的步数内停止的。

主元素问题









Backtracking Algorithm

回溯算法

Following Sections

- 回溯算法的基本思想
- 通过范例学习回溯算法的设计策略
  - （1）装载问题
  - （2）n后问题
  - （3）0-1背包问题
  - （4）旅行售货员问题

回溯算法的基本思想(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。

回溯算法的基本思想(2)

■ 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?

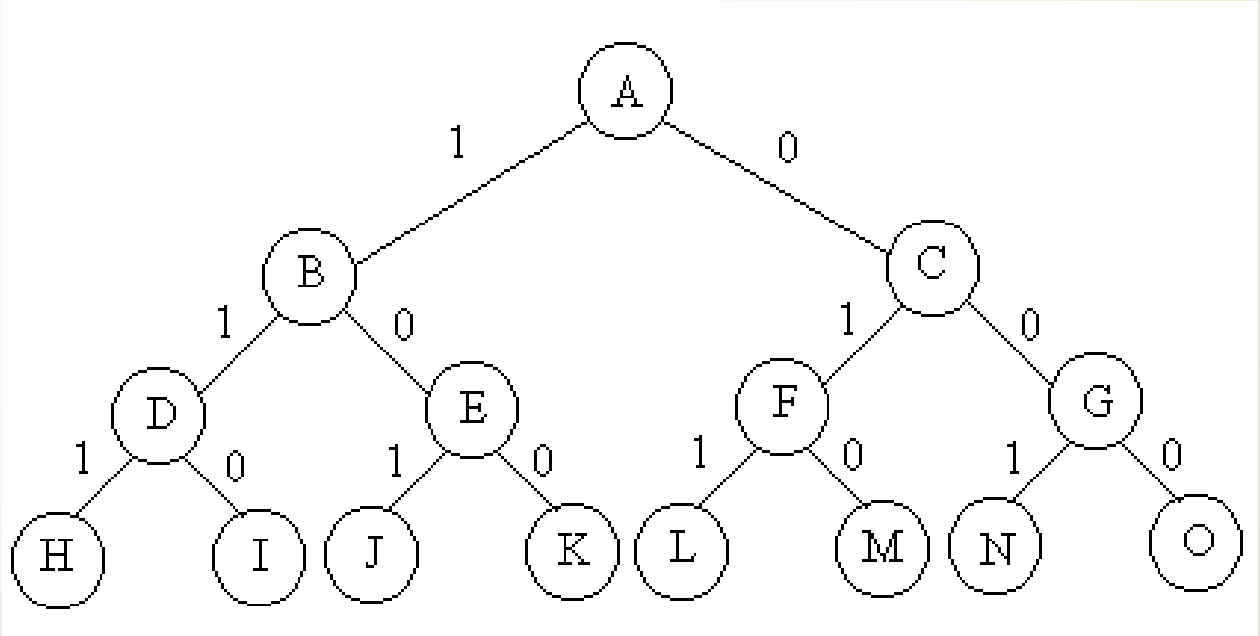
■ 形式化描述如下：给定$c>0, w_i>0, v_i>0, 1\le i\le n$，请找出$n$元0-1向量$(x_1, x_2,\ldots, x_n)$，$x_i \in \{0,1\}$，$1\le i\le n$，使得：

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

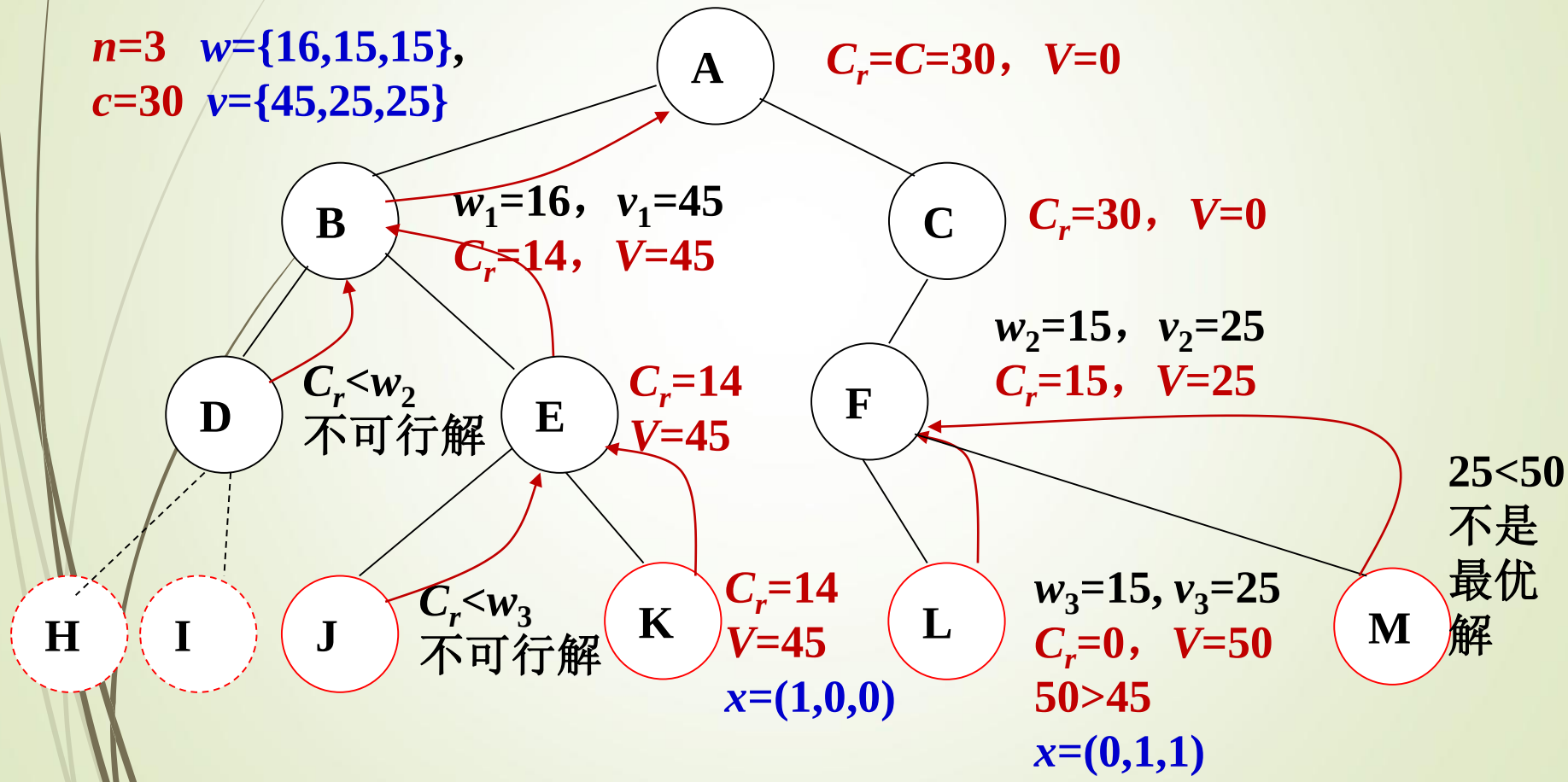
回溯算法的基本思想(3)

- 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



回溯算法的基本思想(4)



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

回溯算法的基本思想(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）。

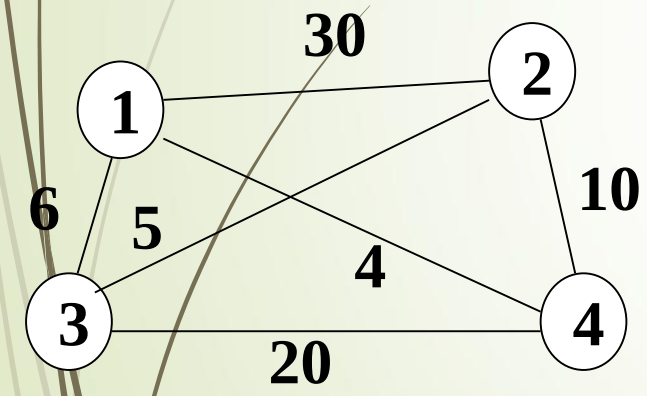
回溯算法的基本思想(6)

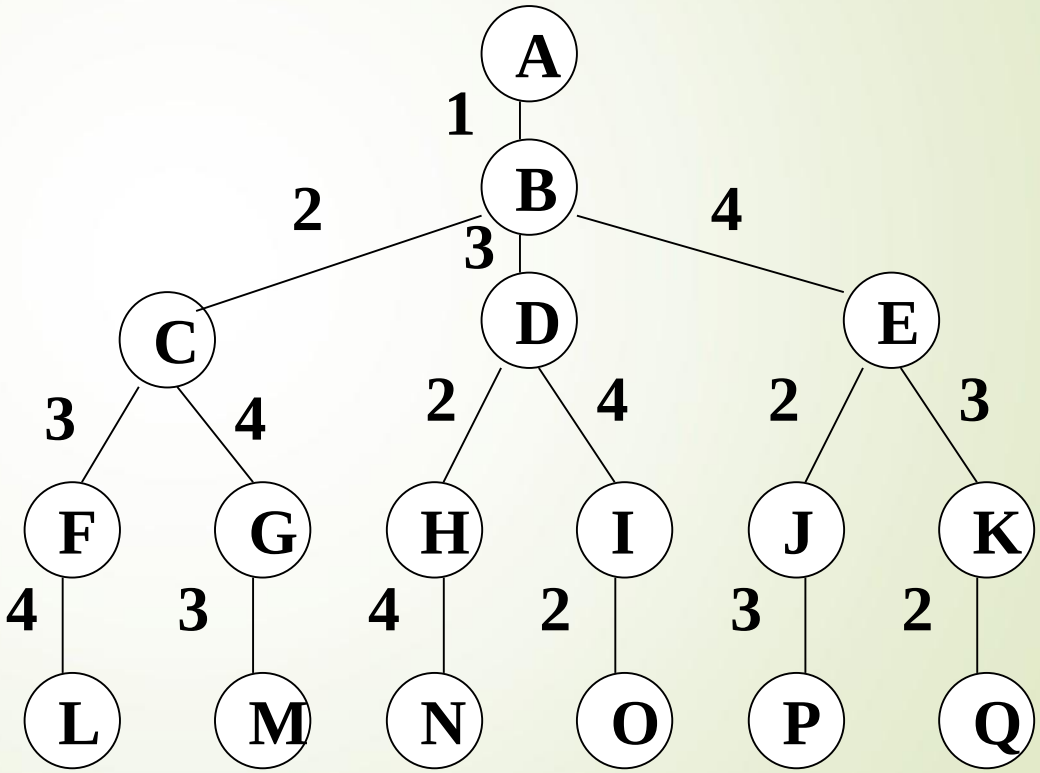
- 旅行售货员(Traveling Salesman Problem)问题：某售货员要到若干城市去推销商品，已知各城市之间的路程，他要选定一条从驻地出发，经过每个城市一遍，最后回到住地的路线，使总的旅费最小。

- 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。

回溯算法的基本思想(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

回溯算法的基本思想(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。

回溯算法的基本思想(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);
          }
}
```

回溯算法的基本思想(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--;
    }
}
```

Following Sections

- 回溯算法的基本思想
- 通过范例学习回溯算法的设计策略
  - （1）装载问题
  - （2）n后问题
  - （3）0-1背包问题
  - （4）旅行售货员问题

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批集装箱装上这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)

算法的复杂性为：$O(2^n)$

■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
}
```

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;
}
```

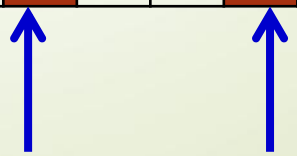
装载问题(6)

- **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)

算法的复杂性依然为：$O(2^n)$

- 优化的核心函数**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
    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];
}
```

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];
    BackTrack( i + 1 ) ;
    cw -= w[i];}
    if (cw + r > bestw) //尝试x[i] = 0
        BackTrack( i + 1 ) ;
    r += w[i];
}
```

装载问题(11)

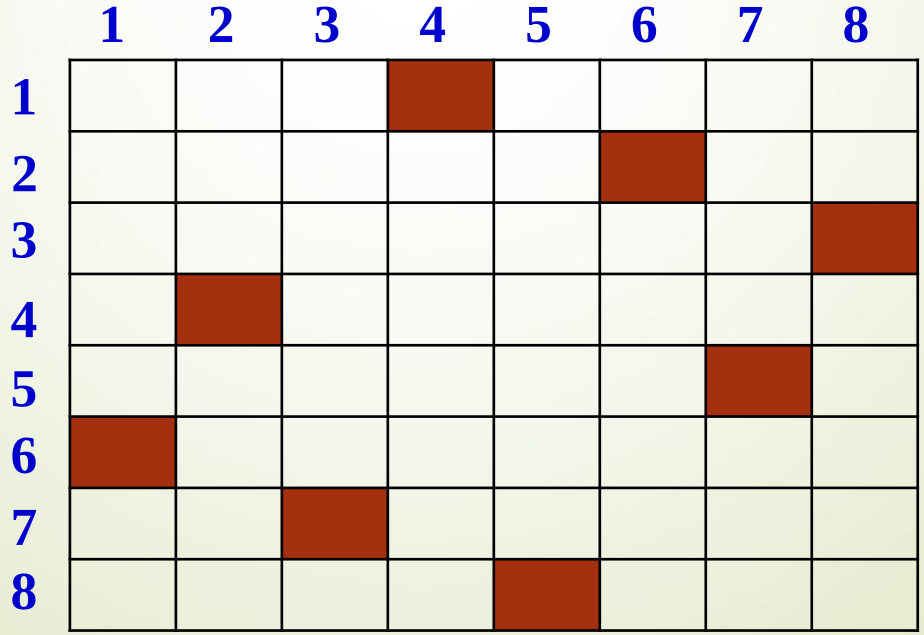
■ 迭代回溯：因为数组x 中记录可在树中移动的所有路径，故可以消除大小为n的递归栈空间。

```
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 + + ;
    }
  }
}
```

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。



K=0

K=1

K=2

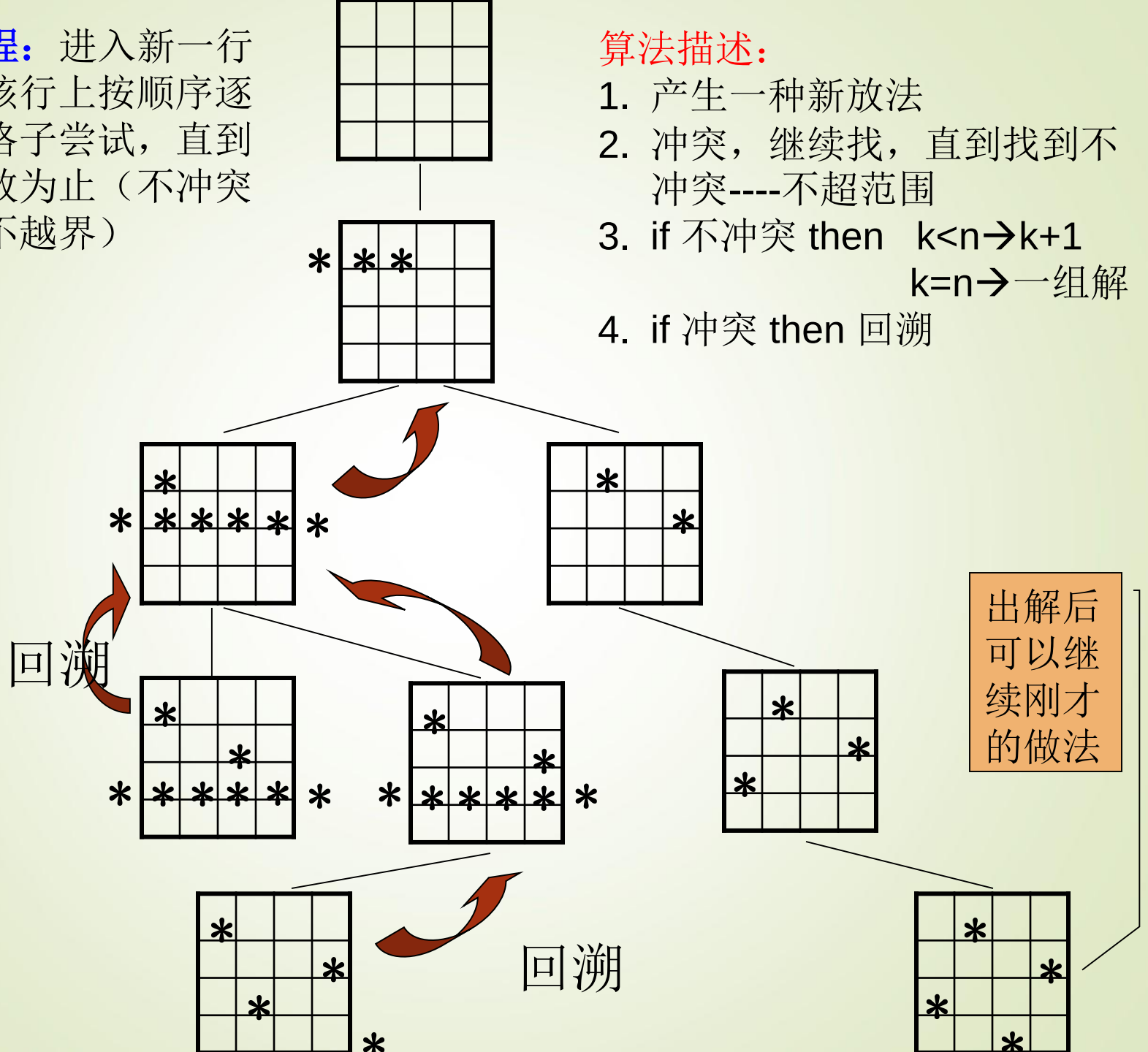
K=3

K=4

过程：进入新一行，该行上按顺序逐个格子尝试，直到能放为止（不冲突、不越界）

算法描述：

1. 产生一种新放法
2. 冲突，继续找，直到找到不冲突----不超范围
3. if 不冲突 then　k<n→k+1
   k=n→一组解
4. if 冲突 then 回溯



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皇后问题的拉斯维加斯算法

对于N皇问题的任何一个解而言，每一个皇后在棋盘上的位置无任何规律，不具有系统性，而更象是随机放置的。由此容易想到拉斯维加斯算法。

在棋盘上相继的各行中随机地放置皇后，并注意使新放置的皇后与已放置的皇后互不攻击，直至n个皇后均已相容地放置好，或已没有下一个皇后的可放置位置时为止。

7.4 N皇后问题

如果将上述随机放置策略与回溯法相结合，可能会获得更好的效果。

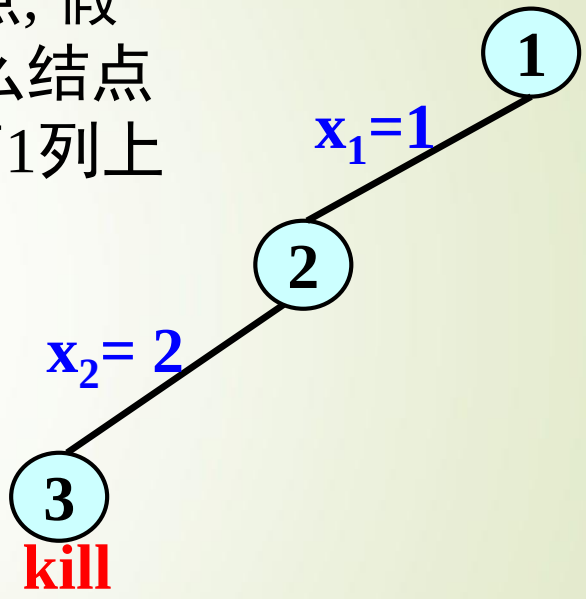
可以先在棋盘的若干行中随机地放置皇后，然后在后继行中用回溯法继续放置，直至找到一个解或宣告失败。

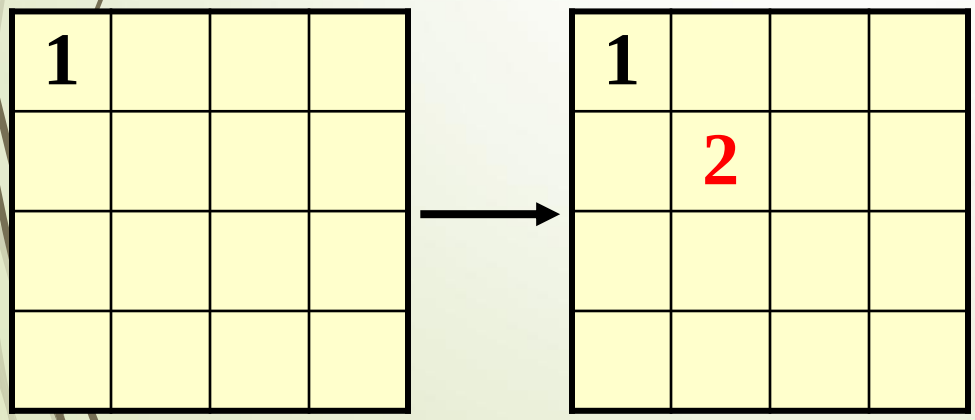
随机放置的皇后越多，后继回溯搜索所需的时间就越少，但失败的概率也就越大。

7.4 N皇后问题的回溯解法

- 开始把根结点作为唯一的活结点, 根结点就成为扩展结点而且路径为( ); 接着生成子结点, 假设按自然数递增的次序来生成子结点, 那么结点2被生成, 这条路径为(1), 即把皇后1放在第1列上

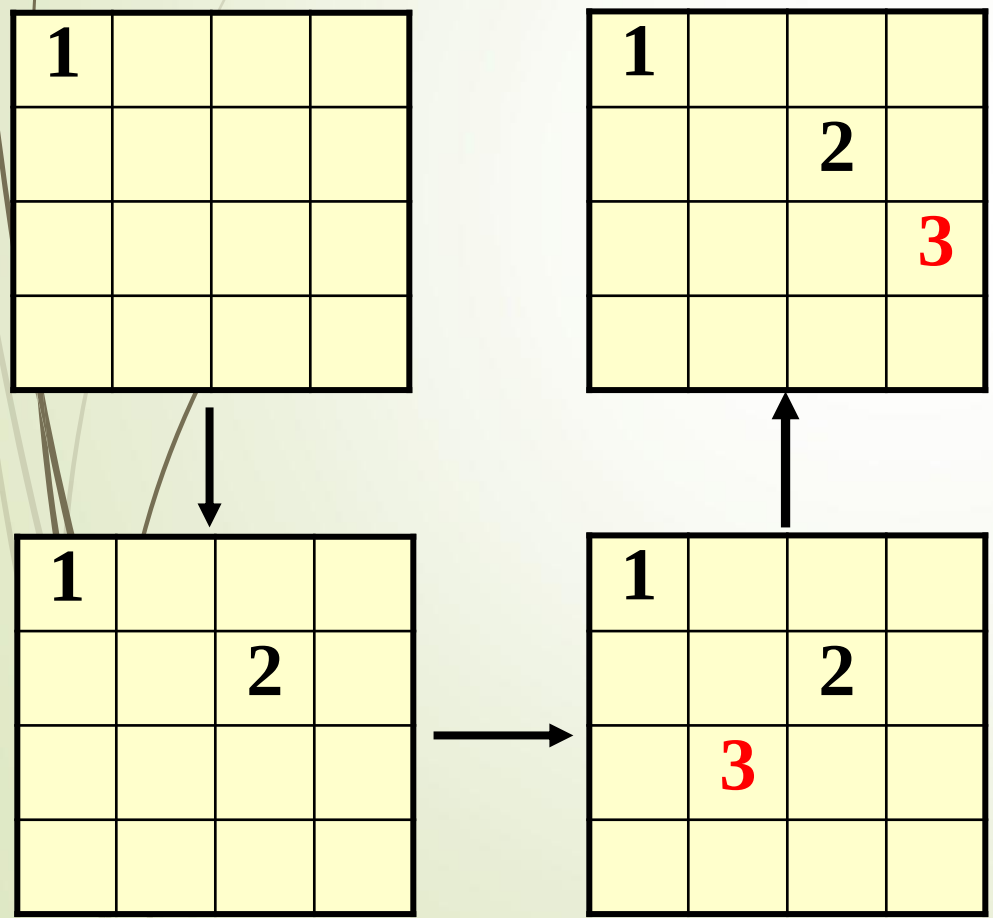
- 结点2变成扩展结点, 它再生成结点3, 路径变为(1, 2), 即皇后1在第1列上,王后2在第2列上, 所以结点3被杀死, 此时应回溯.

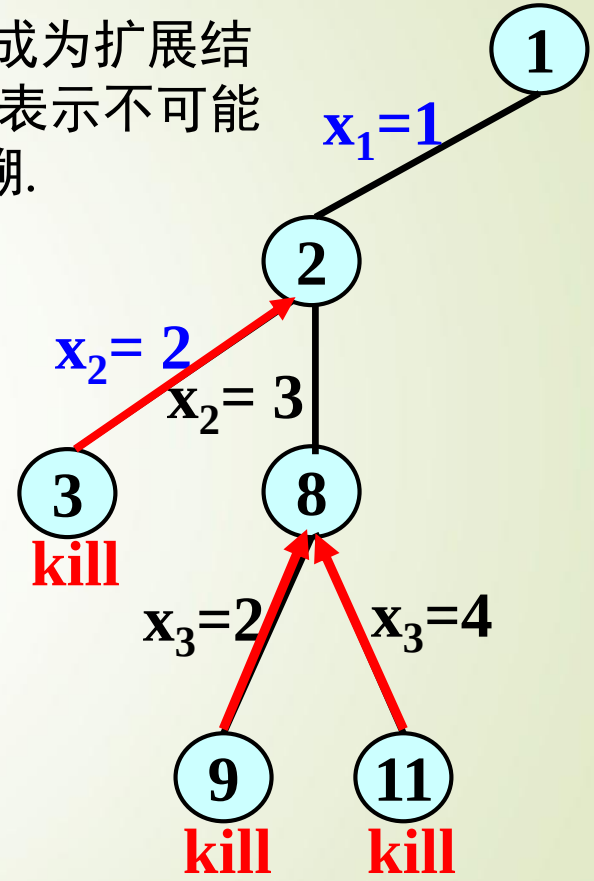




7.4 N皇后问题的回溯解法

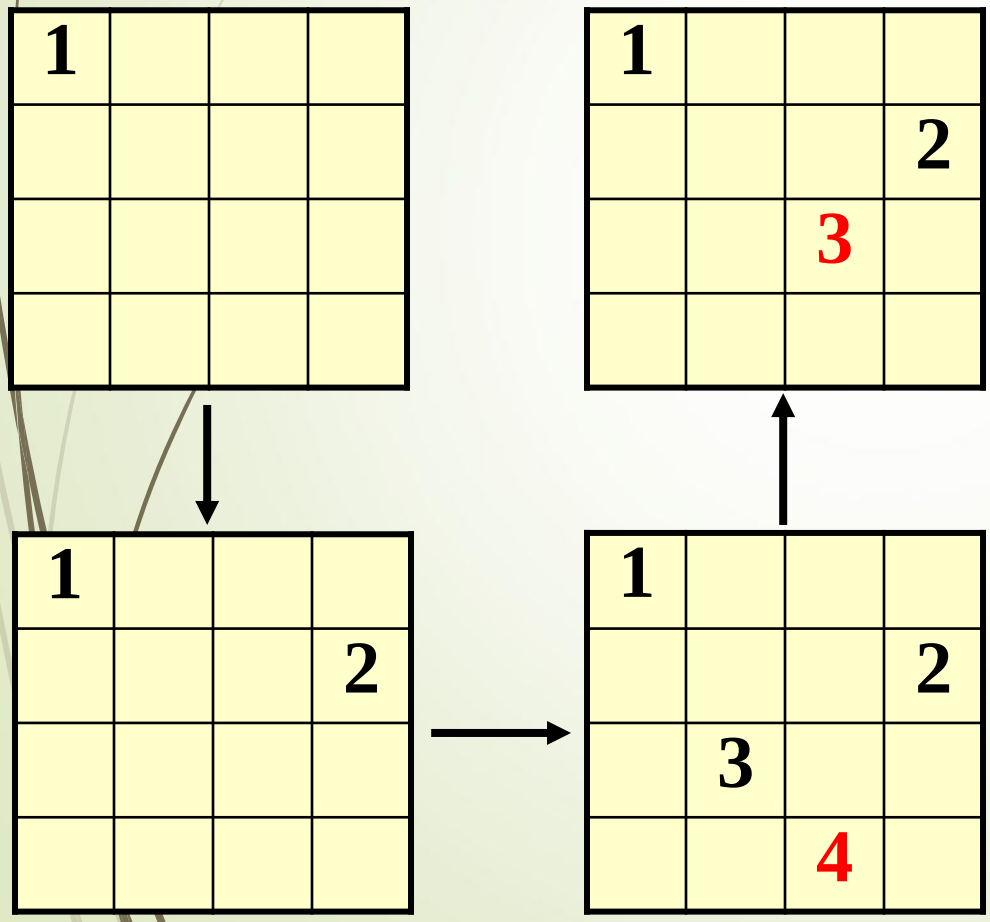
- 回溯到结点2生成结点8, 路径变为(1, 3), 则结点8成为扩展结点, 它生成结点9和结点11都会被杀死(即它的儿子表示不可能导致答案的棋盘格局), 所以结点8也被杀死, 应回溯.

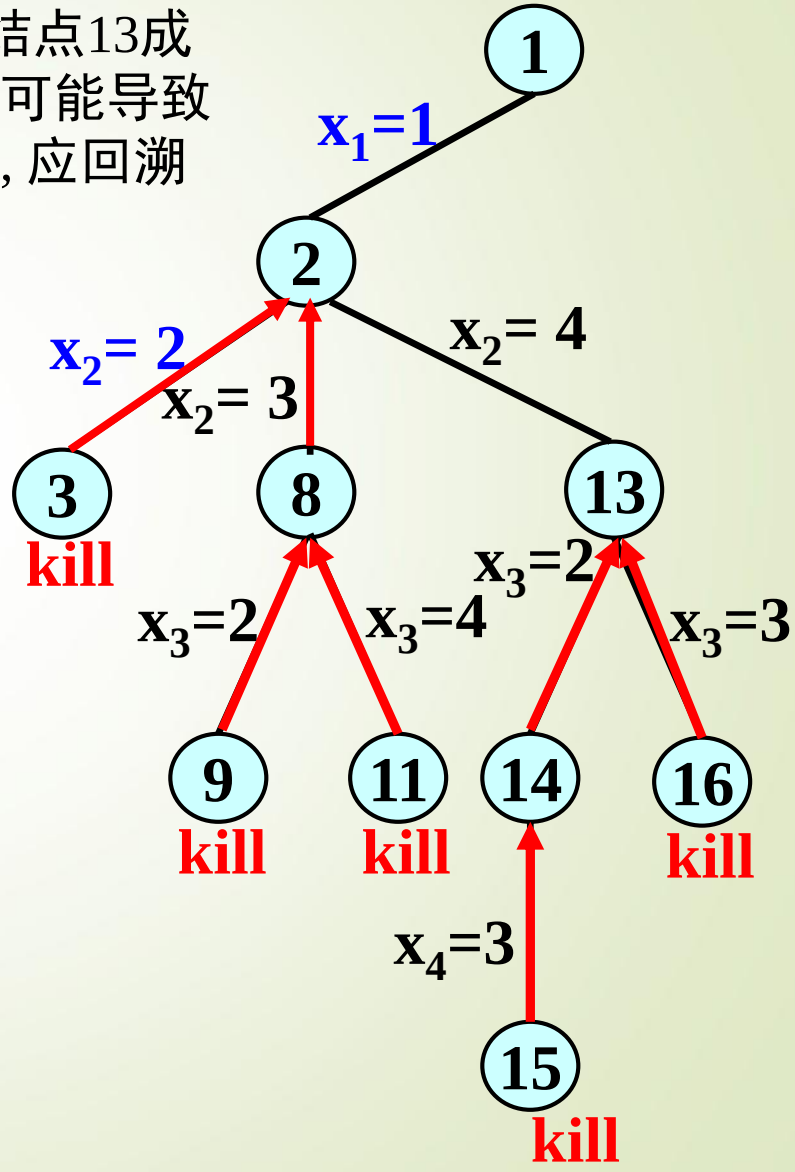




7.4 N皇后问题的回溯解法

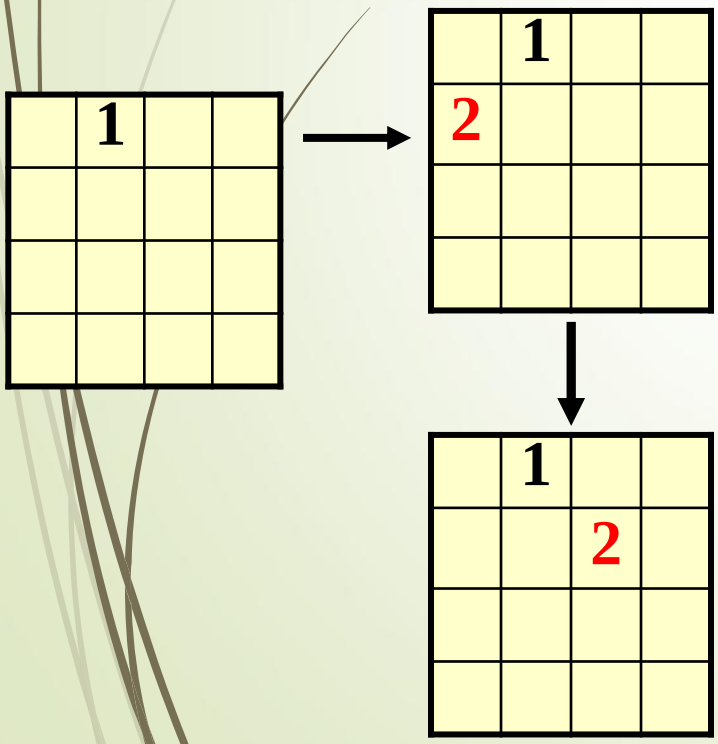
- 回溯到结点2生成结点13, 路径变为(1, 4), 结点13成为扩展结点, 由于它的儿子表示的是一些不可能导致答案结点的棋盘格局, 因此结点13也被杀死, 应回溯

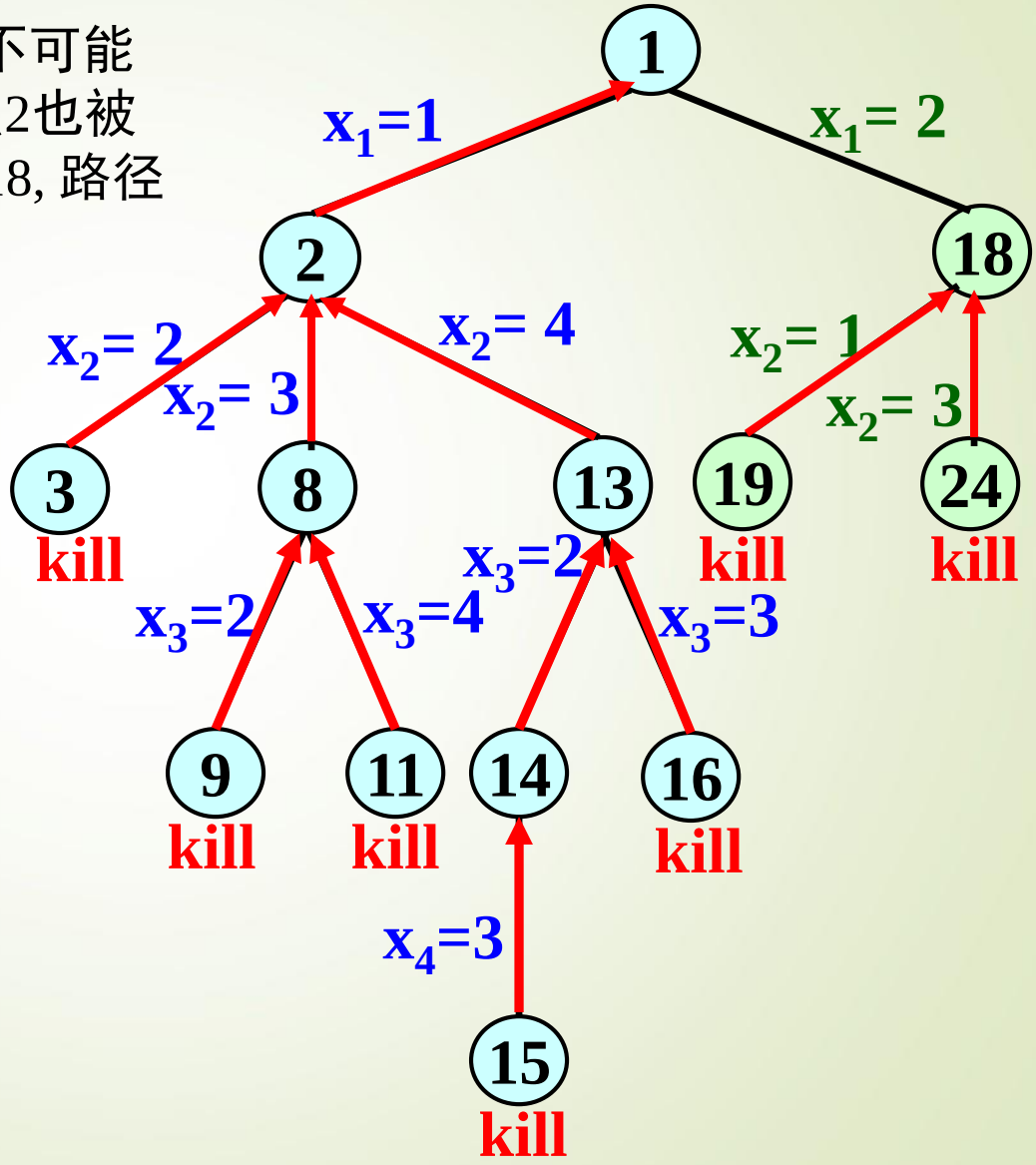




7.4 N皇后问题的回溯解法

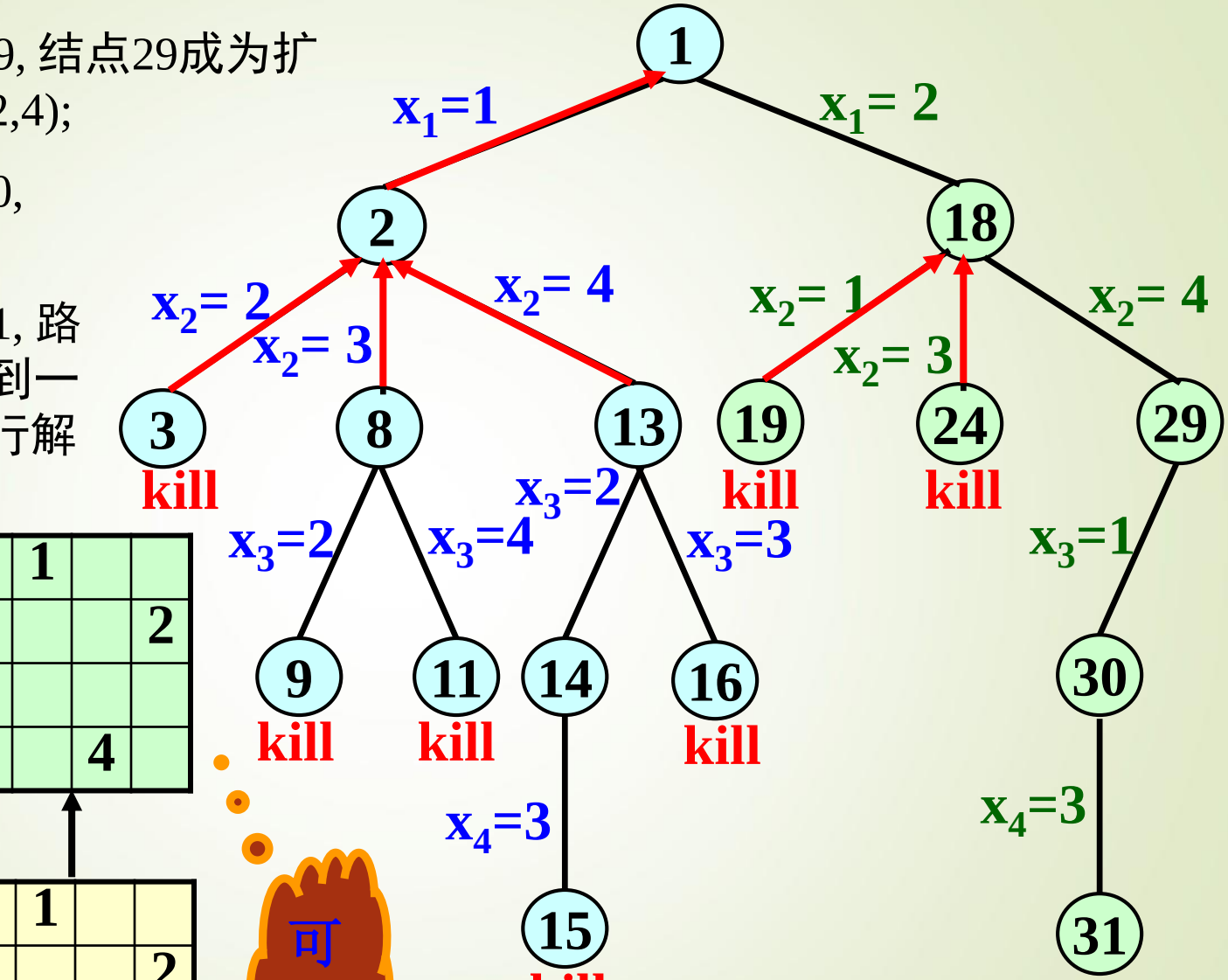
- 结点2的所有儿子表示的都是不可能导致答案的棋盘格局, 因此结点2也被杀死; 再回溯到结点1生成结点18, 路径变为(2).
- 结点18的子结点19、结点24被杀死, 应回溯.





7.4 N皇后问题的回溯解法

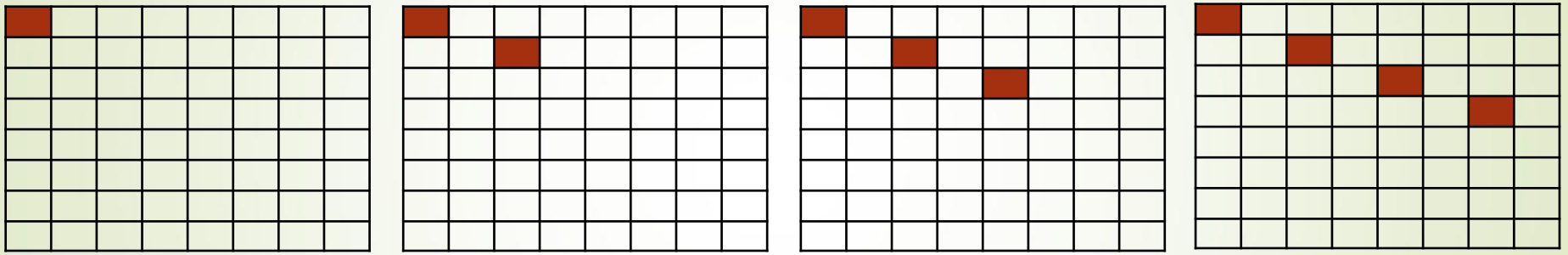
- 结点18生成结点29, 结点29成为扩展结点, 路径变为(2,4);
- 结点29生成结点30, 路径变为(2,4,1)
- 结点30生成结点31, 路径变为(2,4,1,3), 找到一个4-王后问题的可行解



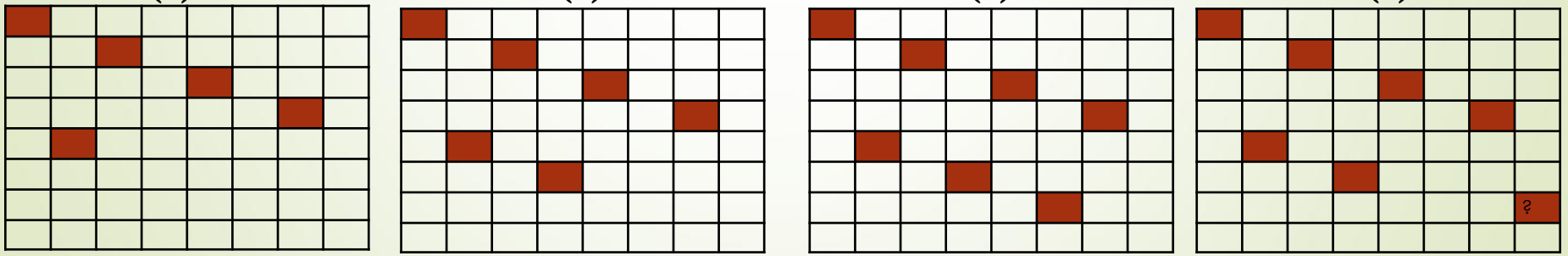
n后问题(3)

- 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)



n后问题(4)

```
bool Queen::Place(int k)
{
        // 测试皇后k置于第x[k]列的合法性
        for(int j = 1; j < k; ++ j)
                if((abs(k-j) == abs(x[j]-x[k])) ||
x[j]==x[k]))
                        return false;
        return true;
}
```

n后问题(5)

```
bool Queen::Backtrack(int t)
{
        // 解n后问题的回溯法
        if(t > n)    //问题的一个可行解
        {
                for(int i=1; i<=n; ++i)
                        y[i] = x[i];  //存储新的解方案
                return true;
        }
        else
                for(int i=1; i<=n; ++i)
                {
                        x[t] = i;
                        if(Place(t) && Backtrack(t+1))
                                return true;
                }
                return false;
}
```

```
bool Queen::QueensLV(int stopVegas)
{
        // 随机放置n个皇后的拉斯维加斯算法
        RandomNumber rnd;    // 随机数产生器
        int k = 1;           // 下一个放置的皇后编号
        int count = 1;
        // 1 <= stopVegas <= n 表示允许随机放置的皇后数
        while((k <= stopVegas) && (count > 0))
        {
                count = 0;
                for(int i = 1; i<=n; ++i)
                {
                        x[k] = i;
                        if(Place(k))
                                y[count++] = i;
                }
                if(count > 0)
                        x[k++] = y[rnd.Random(count)];   // 随机位置
        }
        return (count > 0);   // count > 0表示放置位置成功
}
```

```
bool nQueen(int n)
{
        // 与回溯法结合的解n后问题的拉斯维加斯算法
        Queen X;
        // 初始化X
        X.n = n;
        int *p = new int[n+1];
        int *q = new int[n+1];
        for(int i=0; i<=n; ++i)
        {
                p[i] = 0;
                q[i] = 0;
        }
        X.y = q;
        X.x = p;
        // 设置随机放置皇后的个数
        int stop = 8;
        if(n > 15)
                stop = n-15;
        bool found = false;
        while(! X.QueensLV(stop));

        // 算法的回溯搜索部分
                if(X.Backtrack(stop+1))
                {
                        for(int i=1; i<=n; ++i)
                                cout << p[i] <<
        " ";
                        found = true;
                }
                cout << endl;
                delete [] p;
                delete [] q;
                return found;
```

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：$O(n2^n)$

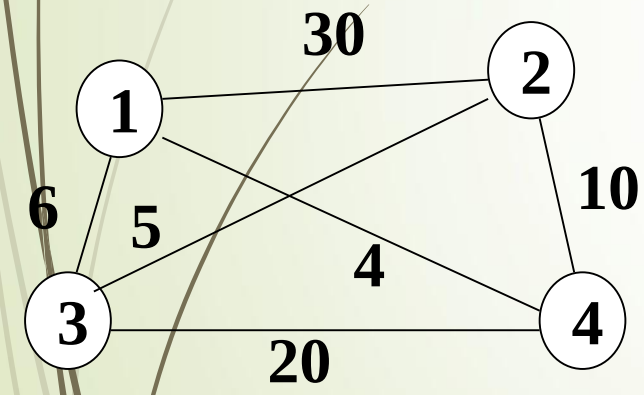
0-1背包问题(2)

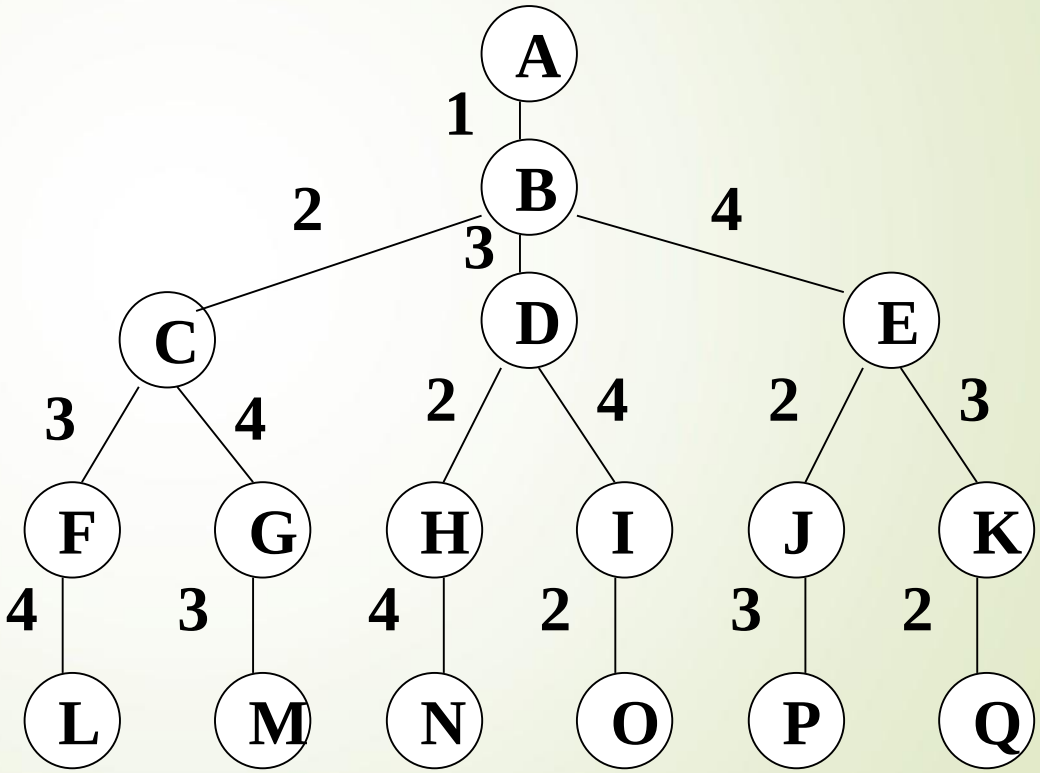
■0-1背包问题的限界函数：首先对物品按单位价值从大到小进行排序，然后按顺序装入物品，具体实现如下：

```
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;
}
```

旅行售货员问题(1)

- 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

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：$O(n!)$

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;
}
```

END