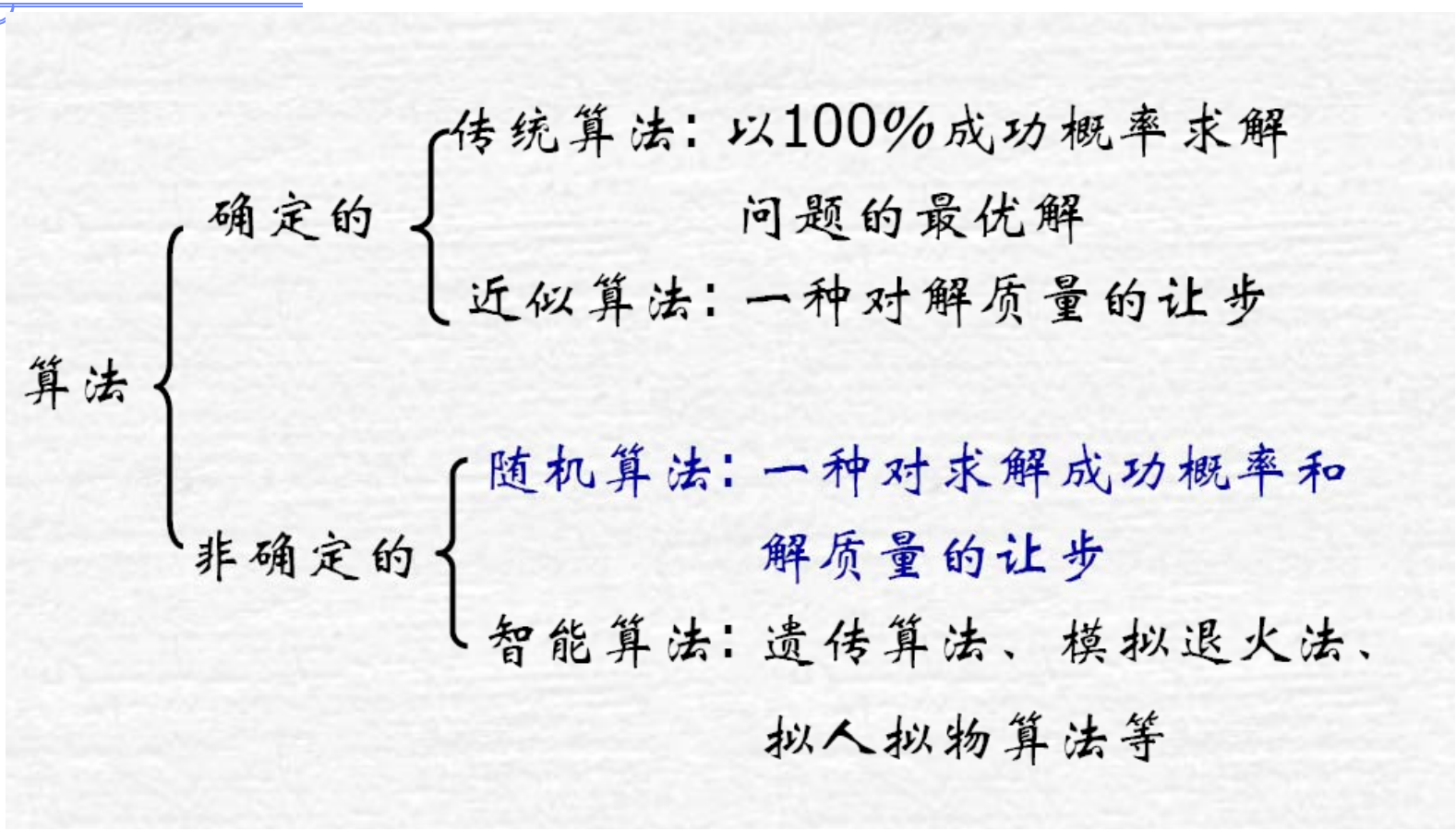
中国科学技术大学

补充4 随机化算法

- 理解产生伪随机数的算法
- 掌握数值随机化算法的设计思想
- 掌握蒙特卡罗算法的设计思想
- 掌握拉斯维加斯算法的设计思想
- 掌握舍伍德算法的设计思想

Sch4-1 方法概述


算法
确定的
传统算法：以100%成功概率求解问题的最优解
近似算法：一种对解质量的让步
非确定的
随机算法：一种对求解成功概率和解质量的让步
智能算法：遗传算法、模拟退火法、拟人拟物算法等

Sch4-1 方法概述

- 定义：是一个概率图灵机。也就是在算法中引入随机因素，即通过随机数选择算法的下一步操作。
- 三要素：输入实例、随机源和停止准则。
- 特点：简单、快速和易于并行化。
- 一种平衡：随机算法可以理解为在时间、空间和随机三大计算资源中的平衡（Lu C.J. 博士论文,1999）
- 重要文献：Motwani R. and Raghavan P., Randomized Algorithms. Cambridge University Press, New York, 1995

Sch4-1 方法概述

- 著名的例子

— Monte Carlo求定积分法
— 随机k-选择算法
— 随机快速排序
— 素性判定的随机算法
— 二阶段随机路由算法

- 重要人物和工作

— De Leeuw等人提出了概率图灵机（1955）
— John Gill的随机算法复杂性理论（1977）
— Rabin的数论和计算几何领域的工作（1976）
— Karp的算法概率分析方法（1985）
— Shor的素因子分解量子算法（1994）

Sch4-1 方法概述

- 常见的随机算法分为4类：

① 数值随机化算法：常用于数值问题的求解，所得到的往往是近似解，解的精度随着计算时间增加而不断提高；

② 蒙特卡罗算法：用于求问题的准确解。该方法可以得到的解，但是该解未必是正确的。求得正确解的概率依赖于算法所用的时间。比较难以判断解是否正确；

③ 拉斯维加斯算法：不会得到不正确的解，但是有时会找不到解。找到正确解的概率随着所用的计算时间的增加而提高。对任一实例，反复调用算法求解足够多次，可使求解失效的概率任意小；

④ 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，消除或者减少这种差别。核心思想是：设法消除最坏情形行为与特定实例之间的关联性。

Sch4-2 随机数

- 随机数在随机化算法设计中扮演着十分重要的角色。在现实计算机上无法产生真正的随机数，因此在随机化算法中使用的随机数都是一定程度上随机的，即伪随机数。
- 线性同余法是产生伪随机数的最常用的方法。由线性同余法产生的随机序列$a_0,a_1,\cdots,a_n$满足

$$\begin{cases} a_0 = d \\ a_n = (ba_{n-1} + c) \bmod m \qquad n = 1,2,\Lambda \end{cases}$$

其中b≥0，c≥0，d≤m。d称为该随机序列的种子。如何选取该方法中的常数b、c和m直接关系到所产生的随机序列的随机性能。这是随机性理论研究的内容，已超出本书讨论的范围。从直观上看，m应取得充分大，因此可取m为机器大数，另外应取gcd(m,b)=1，因此可取b为一素数。

Sch4-2 随机数

- **随机整数算法：**

利用线性同余式计算新的种子，将randSeed右移16位得到一个0~65535间的随机数，然后再将此随机整数映射到0~(n-1)范围内。

```
Random（n, s）
{ //产生 0 : n-1之间的随机整数，s为随机数发生器种子
   if( s==0)
         randSeed= time(0);   //用系统时间产生种子
   else
         randSeed = s;          //由用户提供种子
   randSeed = multiplier * randSeed + adder;
   return  (randSeed >>16) % n ;
}
```

其中： multiplier = 1194211693L， adder = 12345L

Sch4-2 随机数

- 随机实数生成算法

```
fRandom( s )
{  //产生[0，1)之间的随机实数
   return Random( maxshort ) / double( maxshort );
}
```

其中，maxshort = 65536L

思考：如何产生一个符合一定分布的随机数？

Sch4-2 随机数

- 假设抛10次硬币为一个事件，每次抛硬币得到正面和反面是随机的，利用计算机产生的伪随机数模拟抛硬币试验。

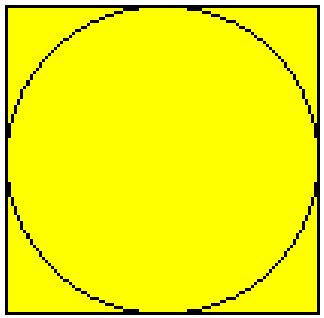
```
int TossCoins( int numberCoins)
{ //随机抛numberCoins次硬币，返回得到正面的次数
    int i, tosses = 0;
    for( i = 0; i<numberCoins; i++ )
    {  //Random(2) = 1表示正面
        tosses += Random(2);
     }
     return tosses;
}

void main()
{  //模拟随机抛硬币试验
   for( j= 0; j<11; j++ )
        heads[j] = 0;          //heads[i]得到i次正面的次数
    for( i = 0; i< 50000; i++ )
        heads[TossCoints(10)] ++ ;
}
```

Sch4-3 数值随机化算法

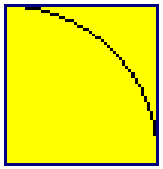
- 例1：用随机投点法计算π值

设有一半径为r的圆及其外切四边形。向该正方形随机地投掷n个点。设落入圆内的点数为k。由于所投入的点在正方形上均匀分布，因而所投入的点落入圆内的概率为 $\frac{\pi r^2}{4r^2}=\frac{\pi}{4}$。所以当n足够大时，k与n之比就逼近这一概率。从而 $\pi \approx \frac{4k}{n}$

```
double Darts(int n)
{// 用随机投点法计算π值
   static RandomNumber dart;
   int k=0;
   for (int i=1;i <=n;i++) {
     double x=dart.fRandom();
     double y=dart.fRandom();
     if ((x*x+y*y)<=1) k++;
     }
   return 4*k/double(n);
}
```



(a)



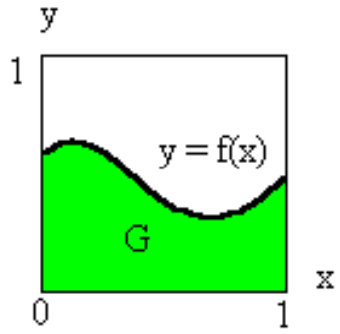
(b)

Sch4-3 数值随机化算法

- 例2：计算定积分

设f(x)是[0，1]上的连续函数，0≤f(x)≤1。需要计算的积分为 $I=\int_0^1 f(x)dx$

积分I等于图中的面积G。



在图所示单位正方形内均匀地作投点试验，则随机点落在曲线下面的概率为

$$P_r\{y \le f(x)\} = \int_0^1 \int_0^{f(x)} dydx = \int_0^1 f(x)dx$$

假设向单位正方形内随机地投入n个点(xi,yi)。如果有m个点落入G内，则随机点落入G内的概率 $I \approx \frac{m}{n}$

Sch4-3 数值随机化算法

```
double  Darts( int n)
{  //用随机投点法计算定积分
   int k=0;
   for( int i = 1; i<= n; i++)
   {
        double   x = dart.fRandom( );
        double   y = dart.fRandom( );
        if( y <= f(x) ) k++;
   }
   return  k/double(n);
}
```

思考：如果被积函数f(x)位于区间[a,b]中有界，并用M, L 表示其最大值和最小值。如何求定积分?

Sch4-3 数值随机化算法

例3：求解下面的非线性方程组

$$\begin{cases} f_1(x_1, x_2, \Lambda, x_n) = 0 \\ f_2(x_1, x_2, \Lambda, x_n) = 0 \\ \Lambda\ \Lambda\ \Lambda\ \Lambda\ \Lambda\ \Lambda\ \Lambda\ \Lambda \\ f_n(x_1, x_2, \Lambda, x_n) = 0 \end{cases}$$

其中，$x_1, x_2, \cdots, x_n$是实变量，f_i是未知量$x_1, x_2, \cdots, x_n$的非线性实函数。要求确定上述方程组在指定求根范围内的一组解。

Sch4-3 数值随机化算法

思路：构造一目标函数：

$$\Phi(x)=\sum f_i^2(x)$$

式中，$x=(x_1,x_2,\cdots,x_n)$。该函数的极小值点就是所求非线性方程组的一组解。

① 直观方法：在指定求根区域内，选定一个x0作为根的初值，按照预先选定的分布，逐个选取随机点x，计算目标函数Φ(x)，并把满足精度要求的随机点x作为所求非线性方程组的近似解。

② 改进方法：在指定求根区域D内，选定一个随机点x0作为随机搜索的出发点。在算法的搜索过程中，假设第j步随机搜索得到的随机搜索点为xj。在第j+1步，计算出下一步的随机搜索增量Δxj。从当前点xj依Δxj得到第j+1步的随机搜索点。当$\Phi(x_{j+1})<\varepsilon$时，取为所求非线性方程组的近似解。否则进行下一步新的随机搜索过程。

Sch4-4 舍伍德算法

设A是一个确定性算法，当它的输入实例为x时所需的计算时间记为$t_{A(x)}$。设X_n是算法A的输入规模为n的实例的全体，则当问题的输入规模为n时，算法A所需的平均时间为

$$\bar{t}_A(n) = \sum_{x \in X_n} t_A(x) / |X_n|$$

这显然不能排除存在$x \in Xn$使得$t_A(x) >> \bar{t}_A(n)$的可能性。希望获得一个随机化算法B，使得对问题的输入规模为n的每一个实例均有

$$t_B(x) = \bar{t}_A(n) + s(n)$$

这就是舍伍德算法设计的基本思想。当$s(n)$与$t_{A(n)}$相比可忽略时，舍伍德算法可获得很好的平均性能。

Sch4-4 舍伍德算法

- 学过的Sherwood算法：快排算法、最坏时间线性选择算法
- 有时也会遇到这样的情况，即所给的确定性算法无法直接改造成舍伍德型算法。此时可借助于随机预处理技术，不改变原有的确定性算法，仅对其输入进行随机洗牌，同样可收到舍伍德算法的效果。例如，对于确定性选择算法，可以用下面的洗牌算法shuffle将数组a中元素随机排列，然后用确定性选择算法求解。这样做所收到的效果与舍伍德型算法的效果是一样的。

```
void Shuffle(Type a[], int n)
{ // 随机洗牌算法
    for (int i=0;i<n;i++)
    {
        int j=rnd.Random(n-i)+i;
        Swap(a[i], a[j]);
    }
}
```

Sch4-4 舍伍德算法

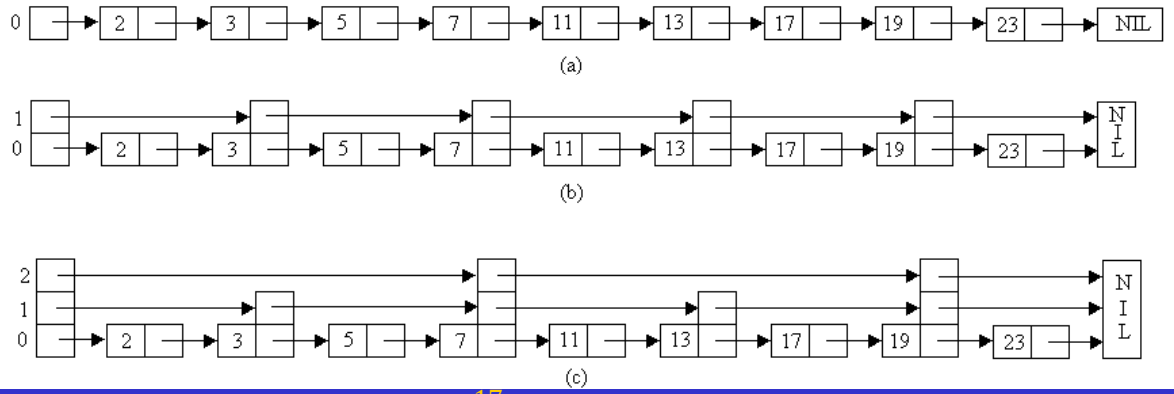
跳跃表：

① 舍伍德型算法的设计思想还可用于设计高效的数据结构；

② 如果用有序链表来表示一个含有n个元素的有序集S，则在最坏情况下，搜索S中一个元素需要Ω(n)计算时间。

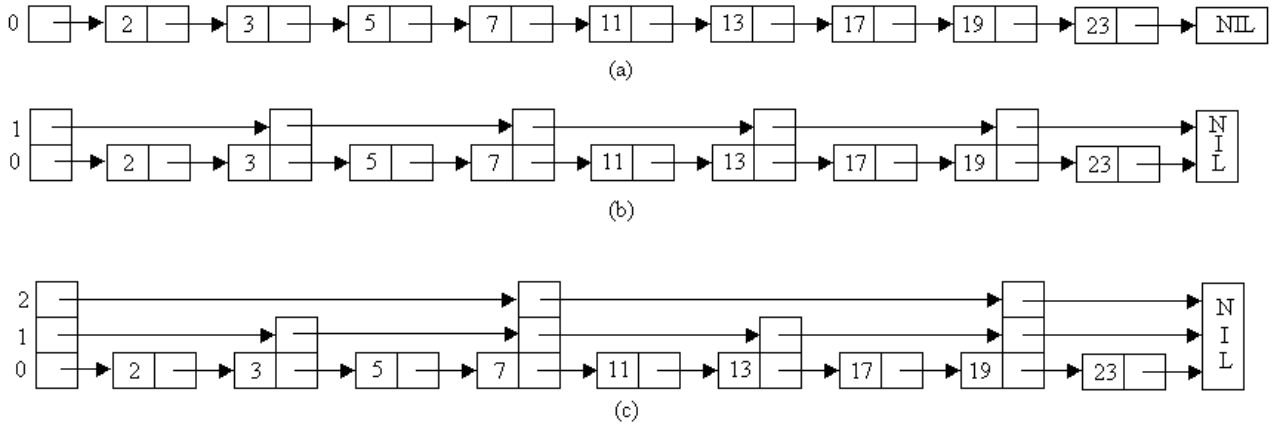
③ 提高有序链表效率的一个技巧是在有序链表的部分结点处增设附加指针以提高其搜索性能。在增设附加指针的有序链表中搜索一个元素时，可借助于附加指针跳过链表中若干结点，加快搜索速度。这种增加了向前附加指针的有序链表称为跳跃表。

④ 应在跳跃表的哪些结点增加附加指针以及在该结点处应增加多少指针完全采用随机化方法来确定。这使得跳跃表可在O(logn)平均时间内支持关于有序集的搜索、插入和删除等运算。



Sch4-4 舍伍德算法

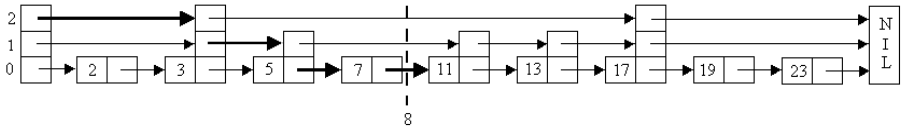
- 在一般情况下，给定一个含有n个元素的有序链表，可以将它改造成一个完全跳跃表，使得每一个k级结点含有k+1个指针，分别跳过2^k-1，$2^{k-1}-1$，…，2^0-1个中间结点。第i个k级结点安排在跳跃表的位置i2k处，i≥0。这样就可以在时间O(logn)内完成集合成员的搜索运算。在一个完全跳跃表中，最高级的结点是⌈logn⌉级结点。

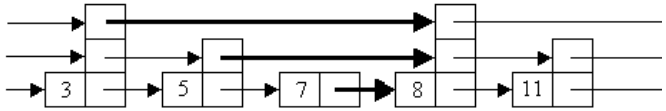


完全跳跃表与完全二叉搜索树的情形非常类似。它虽然可以有效地支持成员搜索运算，但不适应于集合动态变化的情况。集合元素的插入和删除运算会破坏完全跳跃表原有的平衡状态，影响后继元素搜索的效率。

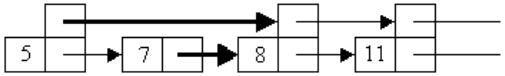
Sch4-4 舍伍德算法

- 为了在动态变化中维持跳跃表中附加指针的平衡性，必须使跳跃表中k级结点数维持在总结点数的一定比例范围内。注意到在一个完全跳跃表中，50%的指针是0级指针；25%的指针是1级指针；…；$(100/2^{k+1})\%$的指针是k级指针。因此，在插入一个元素时，以概率1/2引入一个0级结点，以概率1/4引入一个1级结点，…，以概率$1/2^{k+1}$引入一个k级结点。另一方面，一个i级结点指向下一个同级或更高级的结点，它所跳过的结点数不再准确地维持在2i-1。经过这样的修改，就可以在插入或删除一个元素时，通过对跳跃表的局部修改来维持其平衡性。





(a)



(b)

Sch4-4 舍伍德算法

- 在一个完全跳跃表中，具有i级指针的结点中有一半同时具有i+1级指针。为了维持跳跃表的平衡性，可以事先确定一个实数0<p<1，并要求在跳跃表中维持在具有i级指针的结点中同时具有i+1级指针的结点所占比例约为p。为此目的，在插入一个新结点时，先将其结点级别初始化为0，然后用随机数生成器反复地产生一个[0，1]间的随机实数q。如果q<p，则使新结点级别增加1，直至q≥p。由此产生新结点级别的过程可知，所产生的新结点的级别为0的概率为1-p，级别为1的概率为p(1-p)，…，级别为i的概率为$p^i(1-p)$。如此产生的新结点的级别有可能是一个很大的数，甚至远远超过表中元素的个数。为了避免这种情况，用 $\log_{1/p} n$ 作为新结点级别的上界。其中n是当前跳跃表中结点个数。当前跳跃表中任一结点的级别不超过 $\log_{1/p} n$

Sch4-5 拉斯维加斯(Las Vegas)算法

- 拉斯维加斯算法的一个显著特征是它所做的随机性决策有可能导致算法找不到所需要的解。

```
void obstinate(Object x, Object y)
{// 反复调用拉斯维加斯算法LV(x,y)，直到找到问题的一个解y
  bool success= false;
  while (!success) success=lv(x,y);
}
```

设p(x)是对输入x调用拉斯维加斯算法获得问题的一个解的概率。一个正确的拉斯维加斯算法应该对所有输入x均有p(x)>0。

设t(x)是算法obstinate找到具体实例x的一个解所需的平均时间，s(x)和e(x)分别是算法对于具体实例x求解成功或求解失败所需的平均时间，则有：

$$t(x) = s(x) + \frac{1-p(x)}{p(x)} e(x)$$

Sch4-5 拉斯维加斯(Las Vegas)算法

- 例1：N-皇后问题

分析：对于n后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。由此容易想到下面的拉斯维加斯算法。

思路：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。

Sch4-5 拉斯维加斯(Las Vegas)算法

```
QueensLV( n )
{  //随机放置n个皇后的拉斯维加斯算法
    int  k = 1;   //下一个放置的皇后编号
    int  count = 1;
    while( ( k <= n) && (count > 0) ) {
          count = 0;
           //寻找第i个皇后可能放的所有合法位置
          for ( int i = 1; i <= n; i++){
              x[k] = I;
              if( Place(k) )   y[count++] = i;
          }
          //从合法位置中随机选择一个位置
         if( count > 0 ) x[k++] = y[Random(count)];
    }
    return  (count>0); //放置成功
}
```

```
Place( int k)
{   //判断皇后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;
}
```

Sch4-5 拉斯维加斯(Las Vegas)算法

- 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。

stopVegas	p	s	e	t
0	1.0000	262.00	--	262.00
5	0.5039	33.88	47.23	80.39
12	0.0465	13.00	10.20	222.11

Sch4-5 拉斯维加斯(Las Vegas)算法

```
Backtrack( int t )
{ //解n后问题的回溯法
   if ( t> n )
          return  x;       //找到解
   else
         for( int i=1; i<=n; i++ ){
               x[t] = i;
               if(  Place( t) && Backtrack( t+1 ) )
                        return true;
         }
    return false;
}

//与回溯法相结合的解n后问题的拉斯维加斯算法
void  nQueen( int n)
{
     int stop = 5;   //表示初始随机放置的皇后个数
     if( n >15)  stop  = n-15;
     while( QueensLV( stop ) );
     if ( Backtrack( stop + 1) )   //算法的回溯搜索部分
               打印出解;
 }
```

Sch4-6 蒙特卡罗算法

- 在实际应用中常会遇到一些问题，不论采用确定性算法或随机化算法都无法保证每次都能得到正确的解答。蒙特卡罗算法则在一般情况下可以保证对问题的所有实例都以高概率给出正确解，但是通常无法判定一个具体解是否正确。
- 设p是一个实数，且1/2<p<1。如果一个蒙特卡罗算法对于问题的任一实例得到正确解的概率不小于p，则称该蒙特卡罗算法是p正确的，且称p-1/2是该算法的优势。
- 如果对于同一实例，蒙特卡罗算法不会给出2个不同的正确解答，则称该蒙特卡罗算法是一致的。
- 有些蒙特卡罗算法除了具有描述问题实例的输入参数外，还具有描述错误解可接受概率的参数。这类算法的计算时间复杂性通常由问题的实例规模以及错误解可接受概率的函数来描述。

Sch4-6 蒙特卡罗算法

对于一个一致的p正确蒙特卡罗算法，要提高获得正确解的概率，只要执行该算法若干次，并选择出现频次最高的解即可。

如果重复调用一个一致的(1/2+ε)正确的蒙特卡罗算法2m-1次，得到正确解的概率至少为1-δ，其中，

$$\delta = \frac{1}{2} - \varepsilon \sum_{i=0}^{m-1} \binom{2i}{i} \left(\frac{1}{4} - \varepsilon^2\right)^i \le \frac{(1-4\varepsilon^2)^m}{4\varepsilon\sqrt{\pi m}}$$

对于一个解所给问题的蒙特卡罗算法MC(x)，如果存在问题实例的子集X使得：

(1) 当$x \notin X$时，MC(x)返回的解是正确的；

(2) 当$x \in X$时，正确解是y_0，但MC(x)返回的解未必是y_0。

称上述算法MC(x)是偏y_0的算法。

重复调用一个一致的，p正确偏y_0蒙特卡罗算法k次，可得到一个$O(1-(1-p)^k)$正确的蒙特卡罗算法，且所得算法仍是一个一致的偏y_0蒙特卡罗算法。

Sch4-6 蒙特卡罗算法

- 例1 [主元素问题]：

  设T[1:n]是一个含有n个元素的数组。当|{i|T[i]=x}|>n/2时，称元素x是数组T的主元素。

```
bool Majority(Type *T, int n)
{// 判定所给数组T是否包含主元素
 //的蒙特卡罗算法
   int i=Random(n)+1;
   Type x=T[i];  // 随机选择数组元素
   int k=0;
   for (int j=1;j<=n;j++)
    if (T[j]==x) k++;
   return (k>n/2); // k>n/2 时T含有主元素
}
```

说明：算法对随机选择的数组元素x，测试它是否为数组T的主元素。如果算法返回true，则x肯定是数组T的主元素。反之，返回false，数组T未必没有主元素。可能数组包含主元素，但是元素x不是主元素。由于数组T的非主元素个数小于n/2，故这种情况发生的概率小于1/2，因此上述判定数组T的主元素存在性的算法是一个偏真的1/2正确算法。

Sch4-6 蒙特卡罗算法

- 通过重复调用Majority算法可以把错误概率降低到任何可以接受的范围：

```
bool MajorityMC(Type *T, int n, double e)
{  // 重复调用算法Majority
   int k=ceil(log(1/e)/log(2));
   for (int i=1;i<=k;i++)
     if (Majority(T,n)) return true;
   return false;
}
```

对于任何给定的ε>0，算法majorityMC重复调用⌈log(1/ε)⌉次算法majority。它是一个偏真蒙特卡罗算法，且其错误概率小于ε。算法majorityMC所需的计算时间显然是O(nlog(1/ ε))。

Q & A

Thanks!