第七章

概率算法简介
Probabilistic algorithm

掌握概率算法概念及原理

了解数值随机化算法的设计思想

理解蒙特卡罗算法的设计思想

了解拉斯维加斯算法的设计思想

了解舍伍德算法的设计思想

7.1 基本概念

前面各章讨论的算法的每一个步骤都是确定的，而本章讨论的概率算法允许算法在执行过程中随机地选择一下计算步骤。

- ✓ 概率计算就是在算法中可采用随机选择计算的步骤、元素或参数等。
- ✓ 它的基本特征是计算具有不确定性。
- ✓ 它的解也不一定是最优解。
- ✓ 它在很大程度上能降低算法的复杂度。
- ✓ 在非标准算法中普遍应用概率方法，主要有：

  (1)直接用概率进行数值计算；

  (2)用概率/随机进行选择；

  (3)利用概率加速搜索或避免陷于局部最优。

7.1 基本概念

概率算法:指算法的某些步骤中的某些动作是随机性的.

对所求解问题的同一实例用同一概率算法求解两次所需的时间及所得到的结果可能会有很大的差别。

[算法思路] 求解问题P的概率算法 A'定义如下:

设I是P的一个实例，I的规模| I |=n，在用A'解I的某些时刻，随机地选取变量b的值 (1<=b<=n)，由b的值决定算法的下一步选取，除了选取b的动作外，A'的其它动作是确定的。为简单起见，通常设b的值出现的概率是相等的。

概率算法通常是确定算法随机化后的结果，它没有扩大原有的可计算函数的范围，所以不会影响已有的可计算性概念和结果.

7.1 基本概念

[适用问题]

(1) 对有些问题, 最坏情况是极个别的, 可能耗费很长时间, 而绝大多数情况算法执行效率还是很快的, 这时候, 平均复杂性也许更有意义.

(2) 在许多情况下，当算法在执行过程中面临一个选择时，随机性选择常比最优选择省时。

[设计步骤]

(1) 对给定问题P设计一个确定算法A.

(2) 分析算法A的各个步骤及有关操作，确定随机化的入口和随机化方式,从而将算法A随机化，得到概率算法A'.

(3) 论证A'是一个好的算法，即证明：A'比A更好。

7.1 基本概念

[时间复杂性]

分析算法A’的耗费时, 要确定随机变量的耗费函数，并根据问题的特征和采取的随机模式(通常等概率)估算平均耗费.

平均耗费: 如果对每个实例I∈ P, |I|=n,
算法A’解I的平均耗费<=f(n), 则称A’以平均耗费f(n)解问题p.
其中,平均耗费是指同一实例对A’中所有可能选中的r所耗时间对r的平均.

7.2 概率算法分类

数值概率算法

用于数值问题的求解。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必要的，此时用数值概率算法可得到满意的解。 (投点法求π)

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。 (快速排序,搜索)

7.2 概率算法分类

蒙特卡罗方法

用蒙特卡罗算法能求得问题的一个解，但这个解未必正确。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。一般情况下，无法有效判定所得到的解是否肯定正确。(主元素，阿尔法狗！)

拉斯维加斯算法

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。但有时用拉斯维加斯算法会找不到解。与蒙特卡罗算法类似，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一实例，用同一拉斯维加斯算法反复对该实例求解足够多次，可使求解失效的概率任意小。(n后问题)

7.3 数值概率：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a0,a1,…,an，满足：**(混合同余法)

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

- 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。

7.3 数值概率：随机数

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

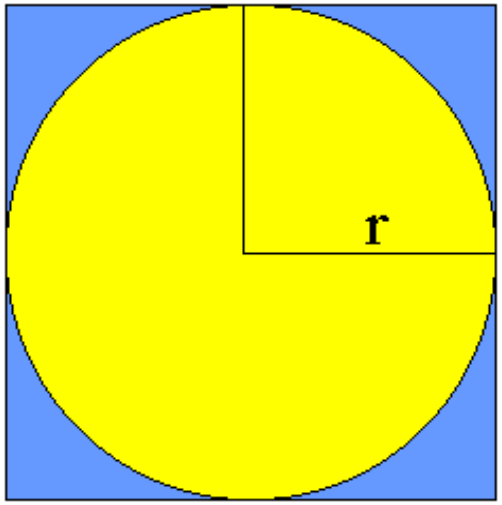
d = 1 种子

m= 11　b=6　c = 0

(1,6,3,7,9,10,5,8,4,2,1,6,3)

7.4 随机投点法计算π值(数值概率算法)

[问题] 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,假定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$\pi r^2/4r^2$, 当n足够大时,k与n之比就逼近这一概率即$\pi/4$, 从而$\pi \approx 4k/n$.

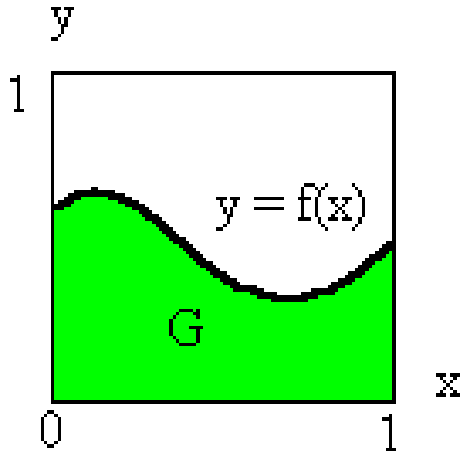



```
Double darts(int n)
    static RandomNumber dart;
    in k=0;
    for (int i=1;i<=n; i++){
      double x=dart.fRandom()
      double y=dart.fRandom()
      if ((x*x+y*y)<=1) k++ }
  return 4*k/double(n)
```

7.4 随机投点法计算定积分(数值概率算法)

用随机投点法计算定积分

设f(x)是[0，1]上的连续函数，且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。



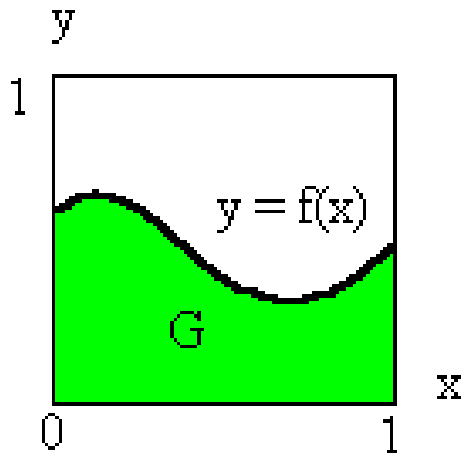
在图所示单位正方形内均匀地作投点试验，则随机点落在曲线下面的概率为：

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

假设向单位正方形内随机地投入n个点(xi，yi)，i = 1,2，…，n。随机点（xi，yi）落入G内，则yi ≤f（xi）。如果有m个点落入G内，则随机点落入G内的概率，即 $I\approx\frac{m}{n}$

7.4 随机投点法计算定积分(数值概率算法)

```
double Darts (int n)
{ // 用随机投点法计算定积分
    static RandomNumber dart;
    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)
}
```



7.5 舍伍德(Sherwood)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∈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。

7.5 舍伍德(Sherwood)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。

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

7.5 舍伍德(Sherwood)算法

随机快速排序算法

快速排序算法：算法的核心在于选择合适的划分基准

```
template <class Type>
void quicksort_random(Type A[],int low,int high)
{
    random_seed(0);              //选择系统当前时间作为随机数种子
    r_quicksort(A,low,high);   // 递归调用随机快速排序算法
}
void r_quicksort(Type A[],int low,int high)
{
    int k;
    if (low<high) {
        k = random(low,high); //产生low到high之间的随机数k
        swap(A[low],A[k]);          //把元素A[k]交换到数组的第一个位置
        k = split(A,low,high);    //按元素A[low]把数组划分为两个
        r_quicksort(A,low,k-1); //排序第一个子数组
        r_quicksort(A,k+1,high);            //排序第二个子数组
    }
}
```

7.6 蒙特卡罗(Monte Carlo)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概率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。

7.6 蒙特卡罗(Monte Carlo)算法　主元素问题

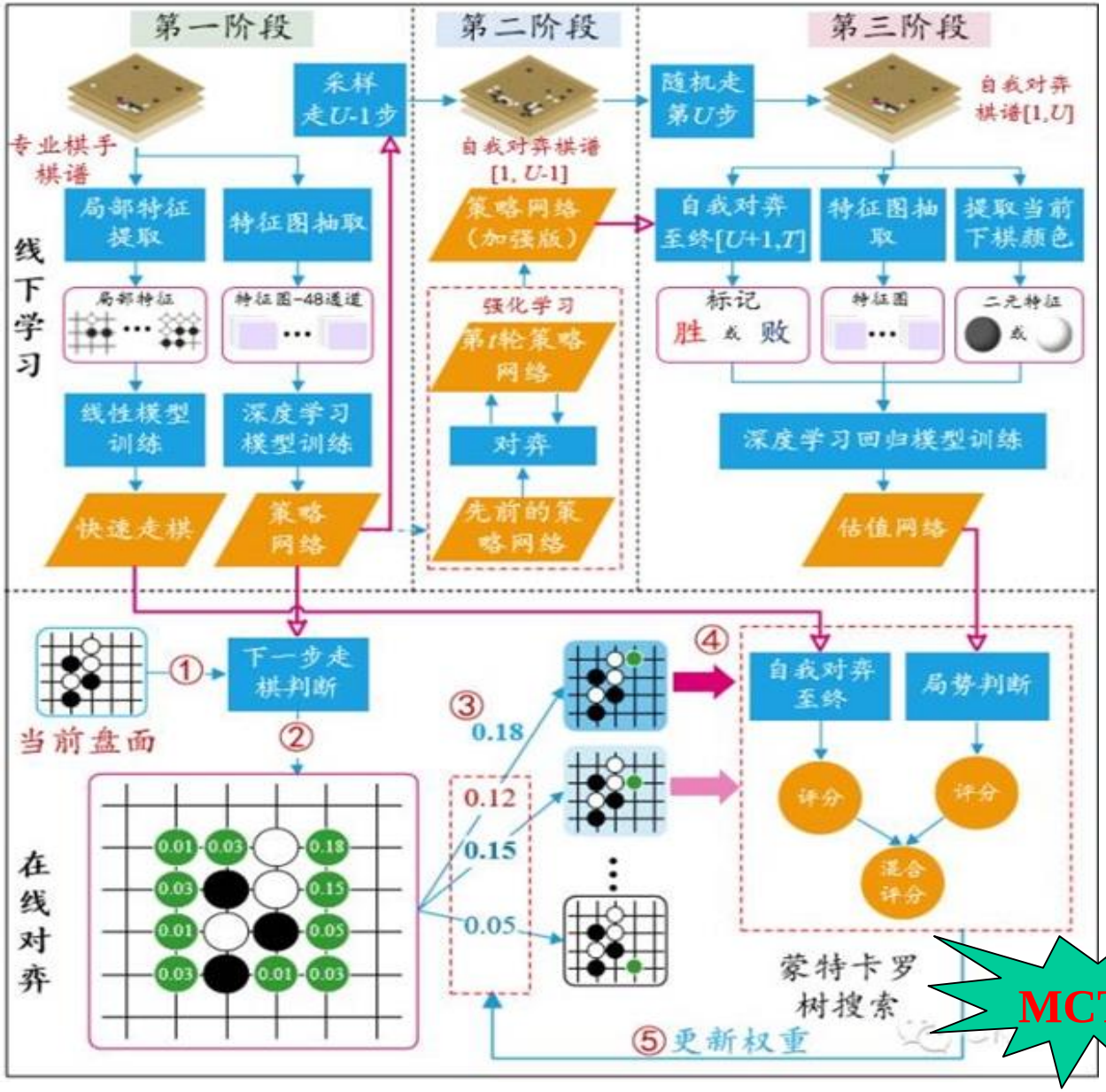
设T[1:n]是一个含有n个元素的数组。当|{i|T[i]=x}|>n/2时，称元素x是数组T的主元素。

```
public static boolean majority(int[]t, int n)
  {// 判定主元素的蒙特卡罗算法
     rnd = new Random();
     int i=rnd.random(n)+1;
     int 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
  }
```

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))。

```
public static boolean majorityMC(int[]t, int n, double e)
  {// 重复é ù次调用算法majority
     int k= (int) Math.ceil(Math.log(1/e)/Math.log(2));
     for (int i=1;i<=k;i++)
       if (majority(t,n)) return true;
     return false;
  }
```

7.6 蒙特卡罗(Monte Carlo)算法　　蒙特卡罗树 AlphaGo

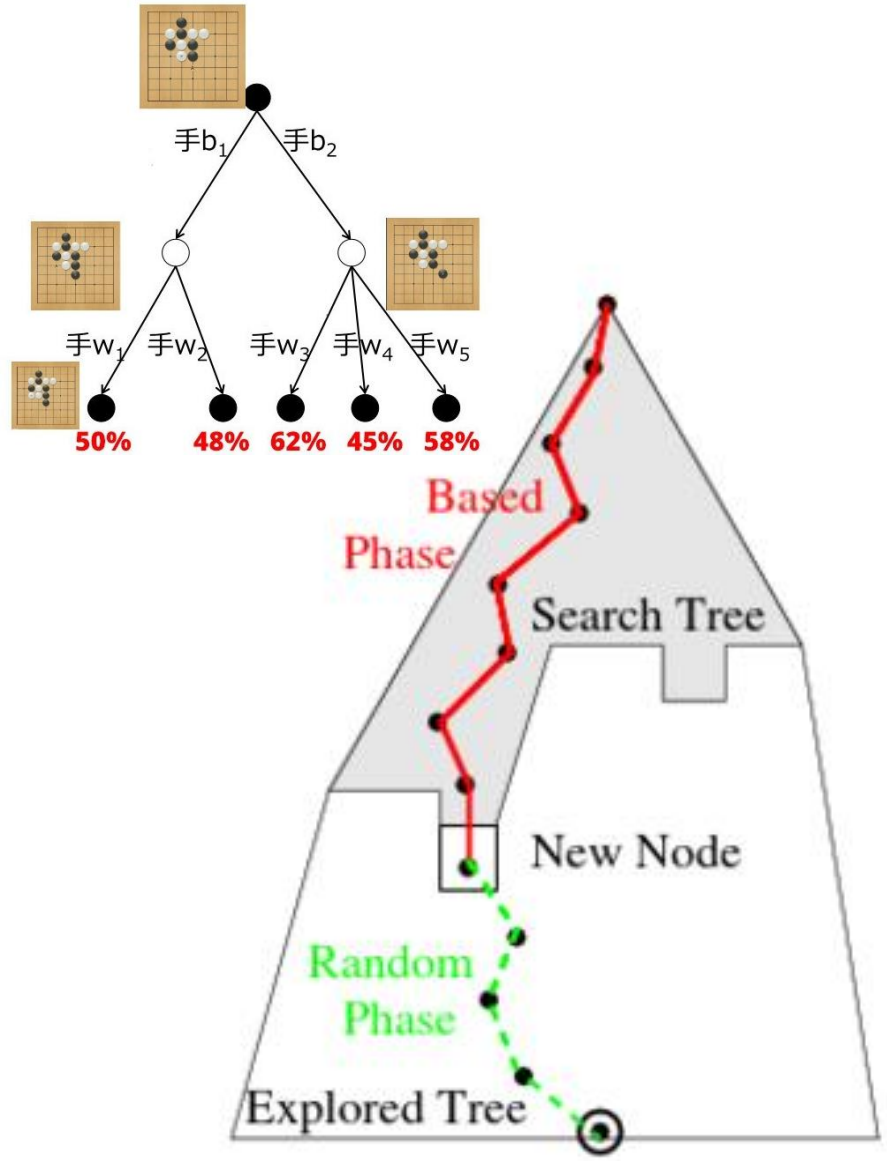


“AlphaGo是一套设计精密的卓越工程，达到了历史性的业界里程碑，这套工程不但有世界顶级的机器学习技术，也有非常高效的代码，并且充分发挥了谷歌在全球最宏伟的计算资源”---李开复

7.6 蒙特卡罗(Monte Carlo)算法

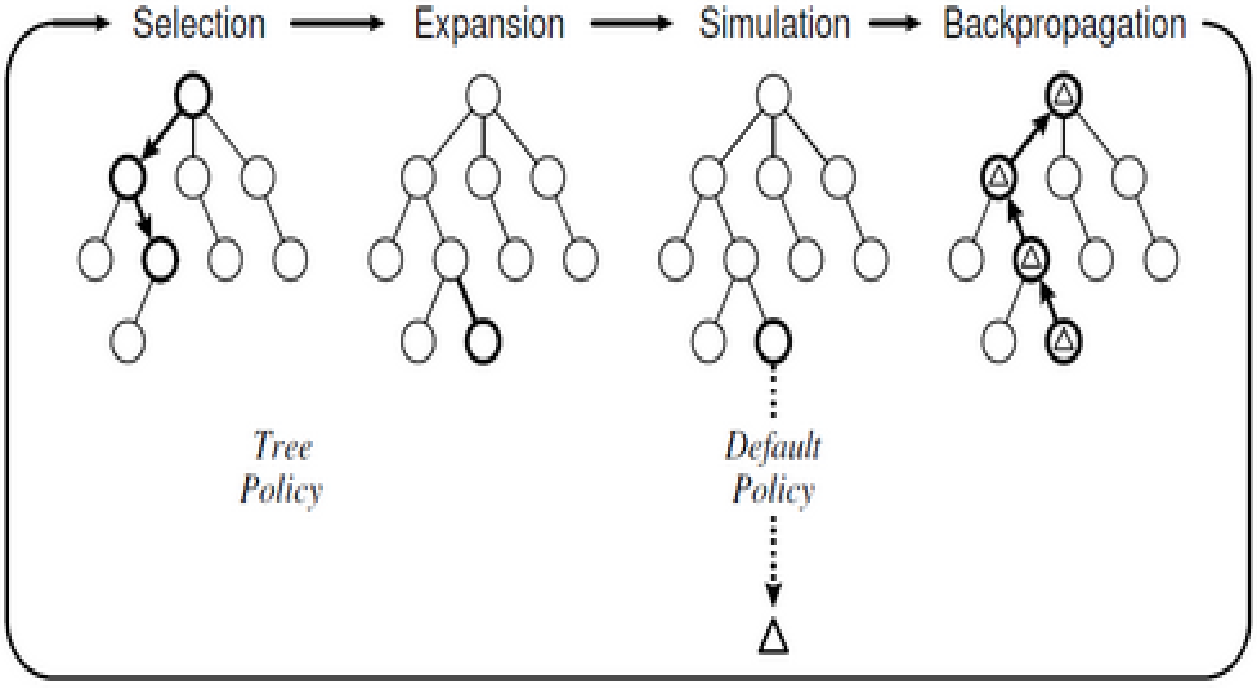
对于一个具体的问题来说，回溯法的有效性往往就体现在当问题实例的规模n较大时，它能够用很少的时间就求出问题的解。而对于一个问题的具体实例我们又很难预测回溯法的算法行为，特别是我们很难估计出回溯法在解这一具体事例时所产生的结点数，这是分析回溯法效率的主要困难。

蒙特卡罗树 AlphaGo



7.6 蒙特卡罗(Monte Carlo)算法　　蒙特卡罗树 MCTS

- Selection ：从root node开始，根据Tree Policy依次选择最佳的child node，直到到达leaf node。Tree Policy就是选择节点（落子）策略，直接影响搜索好坏。
- Expansion：扩展leaf node，将一个或多个可行的落子添加为该leaf node的child node。具体如何暂开，各个程序各有不同。
- Simulation ：根据Default Policy从扩展的位置下棋到终局。完全随机落子就是一种最简单的Default Policy。当然完全随机自然是比较弱的，通过加上一些先验知识等方法改进这一部分能够更加准确的估计落子的价值，增加程序的棋力。
- Backpropagation ：将Simulation的结果沿着传递路径反向传递回root node。




```
function MCTSSEARCH()
    以状态创建根节点;
    while 尚未用完计算时长 do:
        TREEPOLICY();
        DEFAULTPOLICY));
        BACKUP();
    end while
    return (BESTCHILD());
```

7.7 拉斯维加斯(Las Vegas)算法

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。但有时用拉斯维加斯算法找不到解。与蒙特卡罗算法类似，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一实例，用同一拉斯维加斯算法反复对该实例求解足够多次，可使求解失败的概率任意小

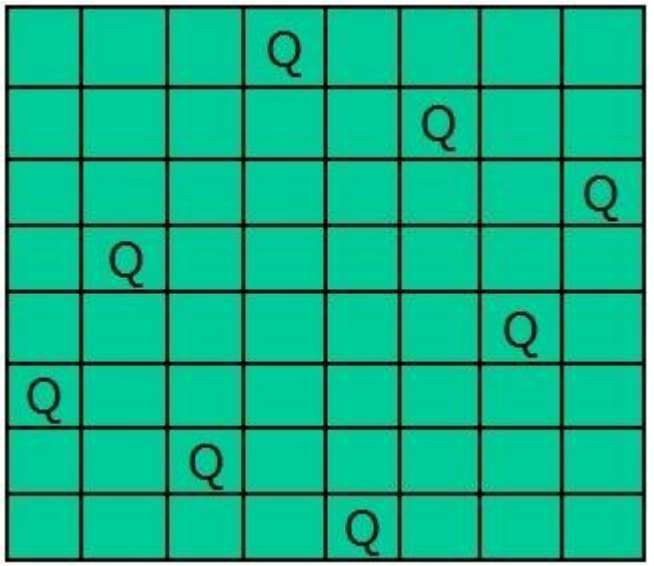
```
void obstinate(Object x, Object y)
  {// 反复调用拉斯维加斯算法LV(x,y)，直到找到问题的一个解y
    bool success= false;
    while (!success) success=lv(x,y);
  }
```

7.7 拉斯维加斯(Las Vegas)算法

N后问题

[问题描述] nXn棋盘上放置n个皇后使得每个皇后互不受攻击. 即任二皇后不能位于同行同列和同一斜线上.

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**。



[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。

7.7 拉斯维加斯(Las Vegas)算法

N后问题

对某行所有列位置进行随机的拉斯维加斯算法

```
bool Queen::QueensLV1(void) //棋盘上随机放置n个皇后拉斯维加斯算法
{
    RandomNumber rnd;      //随机数产生器
    int k=1;               //下一个放置的皇后编号
    int count=maxcout;//尝试产生随机位置的最大次数，用户根据需要设置
    while(k<=n)
     {   int i=0;
        for(i=1;i<=count;i++)
      {
           x[k]=rnd.Random(n)+1;
          if(Place(k))
            break; //第k个皇后在第k行的有效位置存于y数组
          }
       if(i<=count)
          k++;
       else
          break;
    }
    return (k>n);     //k>n表示放置成功
}
```

```
bool Place(int k);  //测试皇后k置于第x[k]列的合法性
int n;              //皇后个数
int *x,*y;          //解向量
```