第七章 随机化算法

课程内容

NP完全性理论与近似算法

随机化算法

线性规划与网络流

递归分治

动态规划

贪心算法

回溯与分支限界

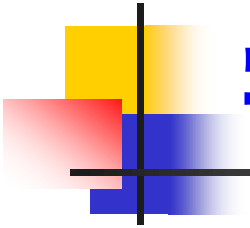
算法分析与问题的计算复杂性

算法高级理论

高级算法

基础算法

算法基础理论



学习要点

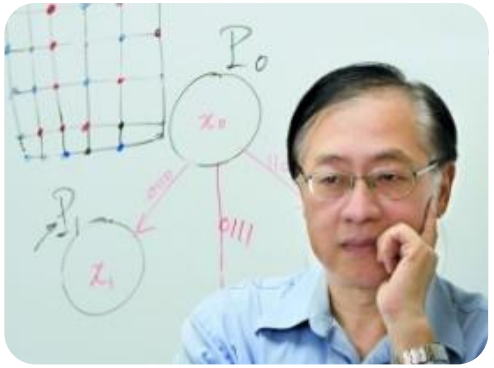
- 理解产生伪随机数的算法
- 掌握数值随机算法的设计思想
- 掌握Las Vegas算法的设计思想
- 掌握Sherwood算法的设计思想
- 掌握Monte Carlo算法的设计思想

随机算法的基本概念

- 例子：
  - 判断函数 $f(x_1,x_2,\ldots x_n)$在区域D中是否恒大于0，f很复杂，不能数学化简，如何判断就很麻烦
  - 若随机产生一个n维坐标$(r_1,r_2,\ldots\ r_n)\in D$，代入得$f(r_1,r_2,\ldots\ r_n)<0$，则可判定区域$D$内$f$不恒大于0
  - 若对很多个随机产生的坐标进行测试，结果次次均大于0，则可说$f>0$的概率是非常大
- 有不少问题，目前只有效率很差的确定性求解算法，但用随机算法去求解，可以很快地获得相当可信的结果

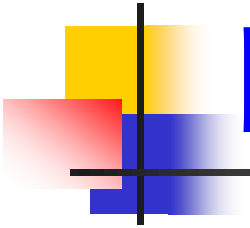
伪随机数的概念

- 伪随机数的生成方法
  - C++：`srand(seed); int x=rand();`
  - Java：`Random`类
  - Python：`import random`
- 真正的随机数
  - 不可预测的，不可见的
- 计算机中的伪随机数
  - 按照一定算法模拟产生的，是确定的，可见的
  - 姚期智（2000年图灵奖，伪随机数生成的贡献）



随机算法的基本概念

- 随机算法
  - 随机算法是一种使用概率和统计方法在其执行过程中对于下一计算步骤作出随机选择的算法
- 随机算法的优越性
  - 对于有些问题：算法简单
  - 对于有些问题：时间复杂性低
  - 对于有些问题：同时兼有简单和时间复杂性低



随机算法四兄弟

1. 随机数值算法
2. Sherwood算法
3. Las Vegas算法
4. Monte Carlo算法

1. 随机数值算法

- 主要用于数值问题求解
- 算法的输出往往是近似解
- 近似解的精确度与算法执行时间成正比

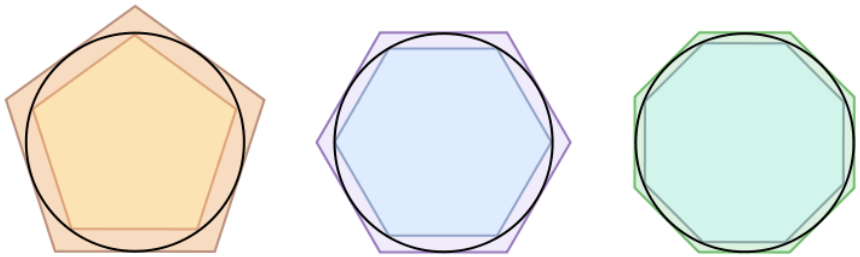
- 应用范例
  - 计算π值
  - 计算定积分
  - 解线性方程组

山巅一寺一壶酒，尔乐苦煞吾，
把酒吃，酒杀尔杀不死，乐尔乐

3.14159265358979323846

祖衝之（四二九—五〇〇），字文遠，範陽遒縣人（今河北涞水縣北）人。南北朝時代著名科學家。他推算出圓周率的數值在3.1415926和3.1415927之間，比歐洲人早一千多年。

祖衝之

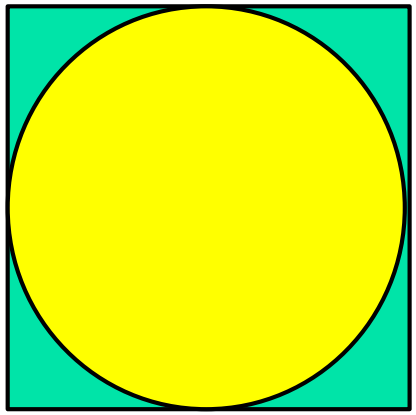


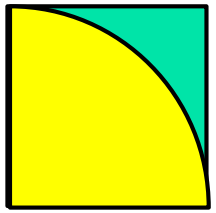
割圆术

用随机数值算法计算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，落入圆内的点数为k
- 当n足够大时，k与n之比接近逼近$\frac{\pi}{4}$，即$\pi \approx \frac{4k}{n}$

```
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);
}
```

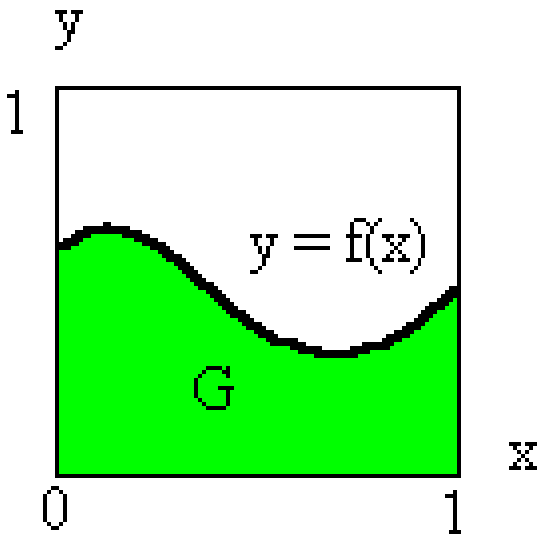




只用计算1/4的部分

计算定积分

- 设$f(x)$是[0, 1]上的连续函数，且$0 \leqslant f(x) \leqslant 1$。
- 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。

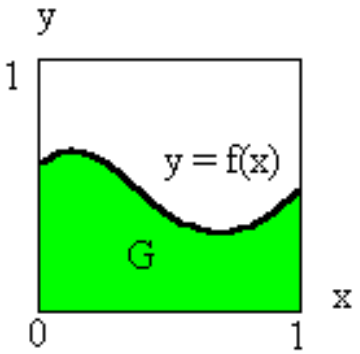


计算定积分

- 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(x_i, y_i)。如果有m个点落入G内，则随机点落入G内的概率 $I \approx \frac{m}{n}$



解非线性方程组

- 求解下面的非线性方程组

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

其中，$x_1,x_2,\ldots,x_n$是实变量，f_i是未知量$x_1,x_2,\ldots,x_n$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。

$$\varphi(x) = \sum_{i=1}^{n} f_i^2(x) \rightarrow \min\{\phi(x)\}$$

解非线性方程组

- 求解下面的非线性方程组

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

在指定求根区域D内，选定一个随机点X_0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X_j。在第j+1步，计算出下一步的随机搜索增量ΔX_j。从当前点X_j依ΔX_j得到第j+1步的随机搜索点。当$\varphi(x)<\varepsilon$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。

2. 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$x \in X_n$使得 $t_A(x) >> \bar{t}_A(n)$ 的可能性。希望获得一个概率算法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Sherwood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。

2. Sherwood算法

- 分析一个算法在平均情况下的计算复杂性，常假定算法的输入服从某一特定的概率分布。
- 例如，数据均匀分布时，快速排序的平均时间为$O(n\log n)$，而输入基本有序时是$O(n^2)$。
- 可用Sherwood算法来消除算法的时间复杂性与输入实例间的某种联系。
- 如果一个确定性算法无法直接改造成Sherwood算法，可用随机预处理技术，不改变原有的确定性算法，仅对其输入实例随机排列（洗牌）。

2. Sherwood算法

- 一定能够求得一个正确解
- 算法最坏与平均复杂性差别大时, 加随机性
- 消除最坏行为与特定实例的联系

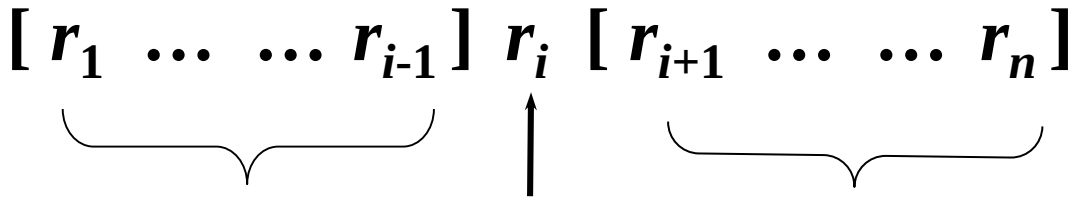
- 应用范例
  - 线性时间选择算法
  - 快速排序算法
  - 搜索有序表
  - 跳跃表

快速排序

快速排序的分治策略是：

- **划分**：选定一个记录作为轴值，以轴值为基准将整个序列划分为两个子序列$r_1 \ldots r_{i-1}$和$r_{i+1} \ldots r_n$，前一个子序列中记录的值均小于或等于轴值，后一个子序列中记录的值均大于或等于轴值；
- **求解子问题**：分别对划分后的每一个子序列递归处理；
- **合并**：由于对子序列$r_1 \ldots r_{i-1}$和$r_{i+1} \ldots r_n$的排序是就地进行的，所以合并不需要执行任何操作。

快速排序



位于最终位置

- 归并排序按照记录在序列中的位置对序列进行划分，
- 快速排序按照记录的值对序列进行划分。

用Sherwood算法优化快速排序

- 基于Sherwood算法的优化
  - 每次随机选一个元素作为划分基准
  - 尽可能使划分基准不会是最大值或最小值

```
int RandomizedPartition (a[], p, r)
{
        int i = Random(p, r);
        Swap(a[i], a[p]);
        return Partition (a, p, r);
}
```

3. Las Vegas算法

- 随机算法运行一次得到正确解或者无解
- 可反复运行LV算法，直到得到正确解为止

```
void obstinate(Object x, Object y)
{   /* 反复调用Las Vegas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);
}
```

LV算法返回为bool型，有两参数：
1)输入；2)成功时保存解。

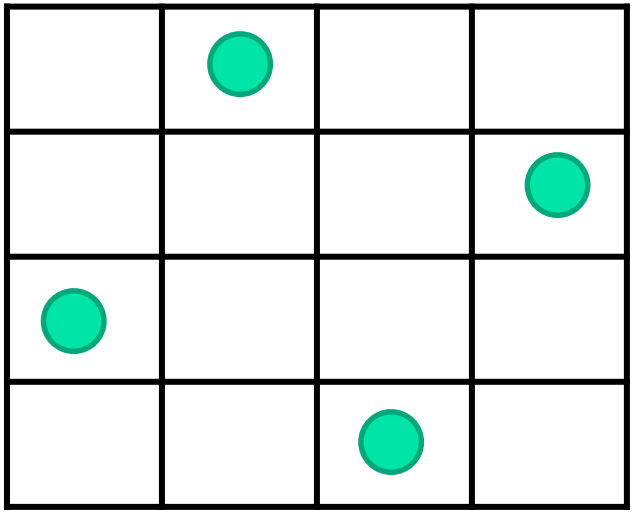
3. Las Vegas算法

- 一旦找到一个解, 该解一定是正确的
- 找到解的概率与算法执行时间成正比
- 增加对问题反复求解次数, 可是求解无效的概率任意小

- 应用范例
  - *n*皇后问题
  - 整数因子分解

用Las Vegas算法求解*n*皇后问题

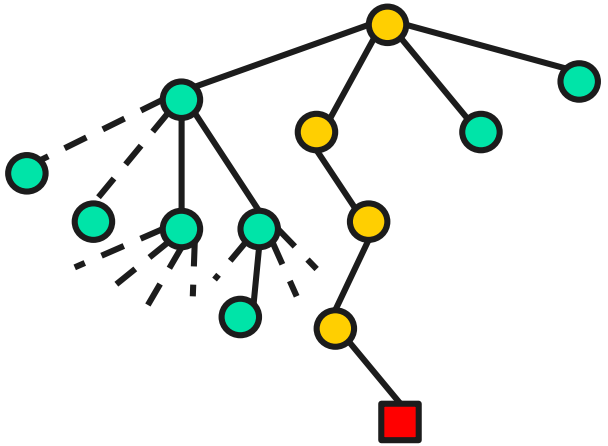
- *n*皇后问题
  - $n \times n$棋盘放*n*个皇后，使皇后之间相互不攻击
  - 皇后位置无任何规律，不具有系统性，而更像是随机放置的
  - 可采用随机算法求解

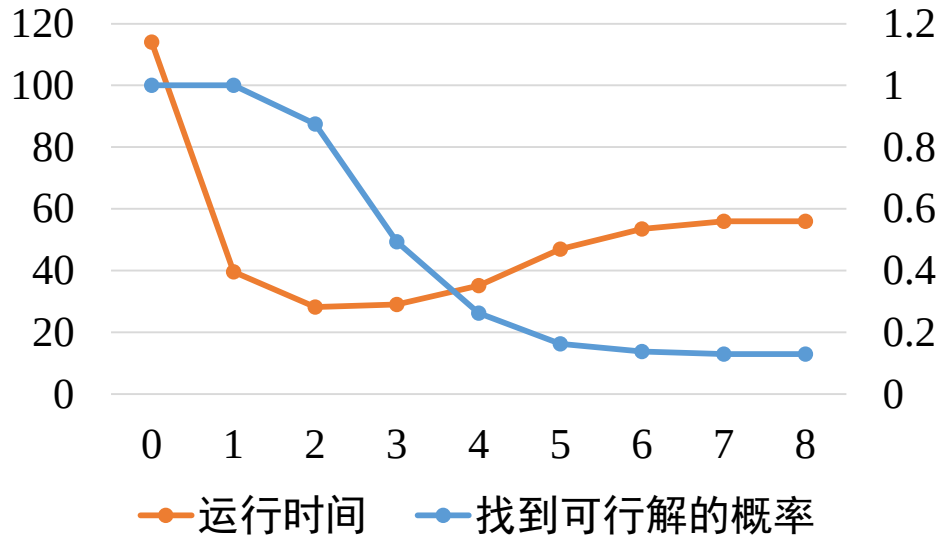


- 基于Las Vegas算法的解法
  - 随机地放置皇后
  - 直至放完*n*个皇后(返回true)，或放不下皇后(返回false)

利用Las Vegas算法优化回溯法

- 例子：n皇后问题
  - 解向量$\langle x_1, x_2, \cdots, x_n \rangle$
  - 先用Las Vegas算法确定前解向量的前k维
  - 再用回溯法求解后n-k维





- 利用Las Vegas算法+回溯法求解8皇后问题
- 当k=3时性能最优

8皇后问题

- 可得：
  1. 随机放两个皇后，再回溯比完全用回溯快大约两倍；
  2. 随机放三个皇后，再回溯比完全用回溯快大约一倍；
  3. 随机放所有皇后，再回溯比完全用回溯慢大约一倍；

不能忽略产生随机数所需的时间，当随机放置所有的皇后时，八皇后问题的求解大约有70%的时间都用在了产生随机数上

4. 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。

- MC算法偶尔会出错，无论任何输入实例，总可以高概率找一正确解。即MC算法总是给出解，此解偶尔可能是不正确，也无法有效判定该解是否正确。

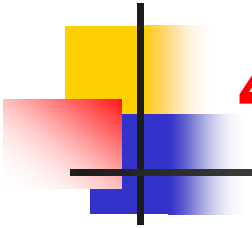
4. Monte Carlo算法

- 例如：设p是一个实数，且$1/2<p<1$。如果一个MC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MC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。
- 如果重复运行一个一致的p正确的MC算法，每次都进行随机选择，即可使产生不正确解的概率→任意小。
- 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。

4. Monte Carlo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$(1/2+\varepsilon)$正确的蒙特卡罗算法$2m-1$次，得到正确解的概率至少为$1-\delta$，其中，

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



4. Monte Carlo算法

- 主要用于求解需要准确解的问题
- 算法可能给出错误解
- 获得精确解概率与算法执行时间成正比

- 应用范例
  - 找主元素
  - 素数测试

主元素问题

- 问题定义
  - 若n元数组T中一半以上元素都是x，则x是T的主元素。
  - 判断数组T是否含有主元素

例如：$T[7]=\{3,2,3,2,3,3,5\}$，主元素为3，含有主元素

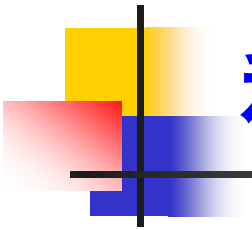
- 暴力解法
  - 统计各元素出现的次数，时间复杂度$O(n^2)$
- 是否还可以改进？

利用Monte Carlo算法判断主元素问题

- **MC算法**
  - **随机选数组中的元素*T*[*i*]统计，如果元素出现次数大于*n*/2，即主元素，返回*true*，否则*false***
- **成功率分析**
  - **如果存在主元素在数组中，那么主元素个数大于*n*/2→ 出错概率小于1/2。**

> 算法以大于1/2概率返回true；
> 算法以小于1/2概率返回false；

如连续运行MC算法*k*次，false减少为2^{-k}，算法发生错误的概率2^{-k}



利用Monte Carlo算法求解主元素问题

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。

```cpp
#include "RandomNumber.h"
#include <iostream>
#include <cmath>
using namespace std;
RandomNumber rnd;
//p正确偏y0蒙特卡洛算法
template <typename Type>
bool Majority(Type *T, int n){
        // 产生0 ~ n-1 之间的随机整数
        int i = rnd.Random(n) + 1;
        Type x = T[i];    // 随机选择数组元素
        int k = 0;
        for(int j = 0; j < n; ++j)
                if(T[j] == x)
                        ++k;
        return (k > n/2);
}
// 重复调用算法Majority
template <typename Type>
bool MajorityMC(Type *T, int n, double e){
        int k = ceil(log(1.0/e) / log(2.0));
        for(int i = 1; i <= k; ++i)
                if(Majority(T, n))
                        return true;
        return false;
}

int
}
```

对于任何给定的$\varepsilon>0$，算法**majorityMC**重复调用$\lceil\log(1/\varepsilon)\rceil$次算法**majority**。它错误概率小于$\varepsilon$。算法**majorityMC**所需的计算时间显然是$O(n\log(1/\varepsilon))$。