随机算法MILLE-RABIN的理论与实验

徐浩博　软件02　2020010108

摘要

随机算法是指在计算中引入随机因素进行计算的算法，它与概率论之间有着密不可分的联系——随机算法的准确性或效率等往往需要借助概率论与数理统计的知识进行证明. 本文中，作者选择了一种素数检测的算法MILLE-RABIN，并且利用概率论的知识论证了它的出错概率是很低的. 最后我们进行了模拟实验，在一定程度上验证了该算法的准确性是很高的.

关键词：随机算法　条件概率　贝叶斯公式　素数检验

MILLE-RABIN算法是用来进行素性检验的算法，由以色列耶路撒冷希伯来大学的Michael O. Rabin教授在卡内基梅隆大学的计算机系教授Gary Lee Miller的确定化素数判定算法基础上提出的随机化算法. 该算法需要运用随机数，并以此来检测某数是不是素数. 此方法以极高的计算效率和较低的出错率而得以广泛应用.

1　MILLE-RABIN算法原理

我们的目标是检测某奇数n是否是素数（偶数的判定是显然的），为此我们假设$n-1=2^tu$，其中u不能被2整除. 由于n-1为偶数，故$t\geq 1$；即将n-1拆成2的幂次和非2的幂次两项因数. 我们可以得到如下分解：

$$\begin{aligned} a^{n-1}-1 &= a^{2^tu}-1 \\ &= (a^{2^{t-1}u}-1)(a^{2^{t-1}u}+1) \\ &= (a^{2^{t-2}u}-1)(a^{2^{t-2}u}+1)(a^{2^{t-1}u}+1) \\ &= \cdots \\ &= (a^u-1)(a^u+1)(a^{2u}+1)\cdots(a^{2^{t-1}u}+1) \end{aligned} \tag{1}$$

由费马小定理：对任何整数a有$a^{n-1}-1\equiv 0(mod\ n)$，结合（1）式，我们有：$(a^u-1)(a^u+1)(a^{2u}+1)\cdots(a^{2^{t-1}u}+1)\equiv 0(mod\ n)$.

定理　若n为质数，且n被分解为$n-1=2^tu$，则对任何正整数a有

$$a^u\equiv 1(mod\ n)\text{或者}a^{2^iu}\equiv -1(mod\ n),\exists i\in\{0,\cdots,t-1\} \tag{2}$$

推论 若存在正整数a满足如下条件，则n为合数：

$$a^u \not\equiv 1(mod\ n)且a^{2^i u} \not\equiv -1(mod\ n), \forall i \in \{0, \cdots, t-1\} \tag{3}$$

由推论，我们只需找出一个不符合要求的a即可确定n为合数，选定一个a后进行条件的判定，我们称之为一次WITNESS，一次WITNESS计算的复杂度为O(logn)，从而大大优化了原始的判断素数算法O($\sqrt{n}$). 然而需要注意的是，推论并不能直接给出素数的判定，为了增加正确性，我们往往需要选取多个a分别进行WITNESS，全部通过后才近似认为是素数.

2 N次WITNESS的出错概率

为了说明MILLE-RABIN的可靠性，我们要推导N次WITNESS后仍判定为素数的出错概率. 为此，我们引入如下定理：

定理 设n为奇数且为合数，则随机选取[2,...,t-1]中的一个数作为a进行WITNESS，那么通过此次WITNESS将n判定为合数的概率为Pr(WITNESS(a)=TRUE)≥3/4. 也即WITNESS未成功的概率为Pr(WITNESS(a)=FALSE)≤1/4.

该定理的证明较为复杂，具体证明过程详见参考文献[1]，除此之外，我们还有素数分布定理$n \to \infty$时，$\pi(x) \sim x/lnx$，即$\pi(x)/x \sim 1/lnx$. 因此，对于任意一个正整数n，我们可以认为它是素数的概率为$Pr(n\ is\ prime) = 1/ln(n)$.

下面让我们来计算N次WITNESS后判定为素数的出错概率，设C_n为n为合数，W_N为经过N次WITNESS仍判定为素数，而我们要求出的概率即为条件概率$P(C_n|W_N)$.

显然，当$n \geq 2$时，$Pr(\overline{C_n}) = Pr(n\ is\ composite)$，而$\overline{C_n}$与$C_n$为对样本空间的一个划分，因此由贝叶斯公式有：

$$Pr(\overline{C_n}|W_N) = \frac{Pr(W_N|\overline{C_n})Pr(\overline{C_n})}{Pr(W_N|\overline{C_n})Pr(\overline{C_n}) + Pr(W_N|C_n)Pr(C_n)} \tag{4}$$

我们来分析右式每一项的值. $Pr(W_N|\overline{C_n})$表示n为素数的条件下经过N次WITNESS仍判定为素数的概率，因此有$Pr(W_N|\overline{C_n}) = 1$；$Pr(\overline{C_n})$表示某正整数n为素数的概率，由素数分布定理，近似有$\Pr(\overline{C_n}) = 1/\ln(n)$，$Pr(C_n) = 1 - 1/\ln(n)$.

$Pr(W_N|C_N)$表示n为合数的条件下经过N次WITNESS仍判定为素数的概率. 首先，我们由定理得知，在n为合数的情况下，一次WITNESS并未判定出n为合数的概率不高于1/4，其次，由于N次WITNESS的a值为随机选取的，彼此独立，因此n次WITNESS之后仍未判定出合数的概率不高于$(1/4)^N$，也即$Pr(W_N|C_n) \leq (1/4)^N$.

综合以上分析，我们继续进行对（4）式的推导：

$$\begin{aligned} Pr(\overline{C_n}|W_N) &= \frac{Pr(W_N|\overline{C_n})Pr(\overline{C_n})}{Pr(W_N|\overline{C_n})Pr(\overline{C_n}) + Pr(W_N|C_n)Pr(C_n)} \\ &\geq \frac{Pr(W_N|\overline{C_n})Pr(\overline{C_n})}{Pr(W_N|\overline{C_n})Pr(\overline{C_n}) + (1/4)^N Pr(C_n)} \\ &= \frac{\frac{1}{\ln n}}{\frac{1}{\ln n} + (\frac{1}{4})^N(1 - \frac{1}{\ln n})} \\ &= \frac{4^N}{4^N + \ln n - 1} \\ &= 1 - \frac{\ln n - 1}{4^N + \ln n - 1} \\ &\geq 1 - \frac{\ln n - 1}{4^N} \\ &\geq 1 - \frac{\ln n}{4^N} \end{aligned} \tag{5}$$

因此有：

$$Pr(C_n|W_N) = 1 - Pr(\overline{C_n}|W_N) \leq \frac{\ln n}{4^N} \tag{6}$$

即当N次WITNESS后未判定为合数的数是素数的概率不大于$\frac{\ln n}{4^N}$.

下面让我们进行一次估算. 我们假设需要判定一个$n \leq 10^8$的数，取$N = 5$即有 $Pr(C_n|W_N) \leq \frac{\ln(10^8)}{4^5} \approx 1\%$，取$N = 18$即有 $Pr(C_n|W_N) \leq \frac{\ln(10^8)}{4^{18}} \approx 10^{-10}$，这说明我们随机进行10次WITNESS就可以保证被判定为素数的n只有小于10^{-10}的概率是合数. 从实际应用角度来说，这一精确度足够满足运算要求.

3 MILLE-RABIN检测实验

下面我们随机抽取$3 \sim 1 \times 10^8$中的一百万个奇数进行实验. 具体来说，我们首先利用埃拉托斯特尼筛法[3]在O(n)的时间内筛出所有1×10^9范围内的素数，然后利用MILLE-RABIN实验，分别选取WITNESS次数N=1,2,5,10,20，并且选取默认随机数引擎进行实验. 其中，我们的错误率定义为抽取的奇数中合数被判定为素数在最后，我们将对比枚举因数的暴力算法与MILL-RABIN算法在时间开销方面的差别.

witness次数	第1次实验		第2次实验		第3次实验		**平均结果**	
	错误率	时间开销	错误率	时间开销	错误率	时间开销	错误率	时间开销
1	6.81×10^{-5}	2.067s	6.60×10^{-5}	2.066s	6.86×10^{-5}	2.112s	6.76×10^{-5}	2.082s
2	9.2×10^{-6}	2.258s	8.8×10^{-6}	2.248s	10.6×10^{-6}	2.319s	9.5×10^{-6}	2.825s
5	3.6×10^{-6}	2.857s	3.6×10^{-6}	2.836s	4.2×10^{-6}	2.840s	3.8×10^{-6}	2.844s
10	1.9×10^{-6}	3.805s	1.6×10^{-6}	3.818s	1.9×10^{-6}	3.805s	1.8×10^{-6}	3.809s

由于实验条件（硬件条件、时间开销、随机数产生算法等）限制，我们仅进行了千万级别的实验，一方面是硬件和时间成本不支持进一步实验，一方面是我们采用c++11的< random>库中默认随机数生成引擎，如果需要更精细的实验结果，则应当首先优化随机数等算法，对于我们的实验来说未免有些本末倒置. 综合以上，我们在有限的条件下进行了较为粗略的实验. 我们看到WITNESS次数等于10时，错误率已经较低，此时错误的次数只有若干次，在此我们停止了实验.

可以通过实验表格发现，当witness次数很小时，我们的错误率达到了很低的水平，例如N=5时，平均错误率已经到了10^{-6}数量级. 而根据第2节错误率的估算，错误的概率上界在1%左右. 因此，我们的实际情况要比预期好很多. 然而当WITNESS次数增多，我们的实验值并没有概率的上界理想，这一方面是由于我们的数据规模仅有10^7数量级，在有限的几次实验中很难观察到N=20时的理论概率上界10^{-10}；另外一方面是random的随机数实际上是一定算法下的伪随机数. 为了实验简单起见，我并没有使用时间开销惊人但效果更好的Mersenne Twister等随机数生成引擎.

我们再对比枚举n的所有可能质因数的算法，它计算一百万个奇数需要的时间大约在200s左右，在时间开销方面与MILLE-RABIN产生了较大差距.

4 总结

本篇文章，我们先介绍了一种随机算法——MILLE-RABIN算法的原理，然后从概率论的理论层面上证明了它的出错概率上限是极小的——在一个数是合数的条件下将其误判为质数的的概率小于$\frac{\ln n}{4^N}$，其中n是要判断的数的大小，N是算法中WITNESS的次数. 理论上当N取一个不太小的数时，判断失误的概率也极其小，这一点在我们不太严格的实验中得到了验证. 除此之外，MILLE-RABIN的特性远远超过简单算法，因此在准确率和效率方面，MILLE-RABIN均不失为一种较好的判定素数的算法.

然而本实验也存在诸多问题，如理论的错误概率上限在实验中并未显现出来，这与我们的设备以及实验的数据规模等有关. 未来进一步实验仍可以在这方面进行改进和提高.

附录 1　参考文献

[1] Conrad, K. (2011). The Miller–Rabin Test. Encyclopedia of Cryptography and Security.

[2] Cormen, T., Leiserson, C., Rivest, R., Stein, C., & EbscoHost. (2009). Introduction to algorithms (3rd ed.). Cambridge, Mass.: MIT Press, 965-975.

[3] Horsley, S. (1772). Being an account of his method of finding all the Prime numbers, by the Rev. Samuel Horsley, FR S. Philosophical Transactions of the Royal Society of London, (62), 327-347.

附录 2　实验代码与环境

操作系统：Windows 10
编译器：g++ (gcc 6.3.0)
处理器：Intel Core i7-10750H 六核CPU @ 2.60GHz
编程语言：C++11

```
#include <random>
#include <cstdio>
#include <cstdlib>
#include <algorithm>
#include <ctime>
using namespace std;

int cnt;
int isComposite[100000005],prime[50000005];

void linear_shaker(int n = 100000000)
{
```

```
13          isComposite[1] = true;
14          for(int i = 2; i <= n; i++)
15          {
16                  if(!isComposite[i])
17                          prime[++cnt] = i;
18                  for(int j = 1; j <= cnt && (i * prime[j]) <= n; j++)
19                  {
20                          isComposite[prime[j] * i] = true;
21                          if(i % prime[j] == 0)break;
22                  }
23          }
24  }
25  int POW(int a, int b, int mod)
26  {
27          a %= mod, b %= mod;
28          int res = 1, now = a, pos = b;
29          while(pos)
30          {
31                  if(pos & 1)
32                          res = (1LL * now * res) % mod;
33                  now = (1LL * now * now) % mod;
34                  pos >>= 1;
35          }
36          return res;
37  }
38  bool brutalJudge(int n)
39  {
40          for(int i = 2; i <= sqrt(n); i++)
41          {
42                  if(n % i == 0) return true;
43          }
44          return false;
45  }
46  bool WITNESS(int a, int n)
```

```
47 {
48         int u = n - 1, t = 0;
49         while(u & 1)
50                 u >>= 1, t++;
51         int a_u = POW(a, u, n);
52         if(a_u % n == 1)
53                 return false;
54         for (int i = 0; i < t; i++)
55         {
56                 if(a_u % n == n - 1)
57                         return false;
58                 a_u = POW(a_u, 2, n);
59         }
60         return true;
61 }
62 bool MilleRabinJudge(int n, int  t)
63 {
64         static int ran = 0;
65         ran++;
66         std::default_random_engine e;
67         std::uniform_int_distribution<int> u(2, n);
68         e.seed(time(0) + ran);
69
70         while(t--)
71                 if(WITNESS(u(e), n))
72                         return true;
73         return false;
74 }
75 int main()
76 {
77         std::default_random_engine e;
78         std::uniform_int_distribution<int> u(1, 49999999);
79         e.seed(time(0) + 123);
80
```

```
81          linear_shaker();
82          clock_t start, finish;
83          start = clock();
84          int err = 0, total = 0;
85          for (int i = 1; i <= 10000000; i++)
86          {
87                  int t = 2 * u(e) + 1;
88
89                  if (isComposite[t])
90                  {
91                          total++;
92                          if (MilleRabinJudge(t, 1) != isComposite[t])
93                                  err++;
94                  }
95          }
96          finish = clock();
97          printf("MILLE-RABIN: %.3f s, wrond answers rate: %llf\n",
98              1.0 * (finish - start) / CLOCKS_PER_SEC, err/total );
99          return 0;
100 }
```