算法设计与分析

2017 年 12 月 30 日

姓名：姜贵平
学号：SA17011142

1 Probabilistic Algorithm

1.1 若将y$\leftarrow uuiform(0,1)$ 改为$y \leftarrow x$, 则上述的算法估计的值是什么?

因为$\frac{k}{n} = \frac{1}{\sqrt{2}}$, 所以结果应该趋于$2\sqrt{2}$

1.2 在在机器上用$4\int_0^1$, 给出不同的n 值其精度

Table 1: Source 1

#include$< stdio.h >$	for(int i=0;$i < s[j]$;i++)
#include$< stdlib.h >$	{
#include$< time.h >$	$x = (float)rand()/RAND\_MAX;$
int main()	$y = (float)rand()/RAND\_MAX);$
{	if($x * x < 1 - y * y$) count++;
float x,y;	}
int s[10]={1e4,1e5,1e6,1e7,1e8,1e9};	printf(”N=%d result is%f”,s[j],4.0*count/s[j]);
for(int j=0;$j < 6$;j++)	return 0;
{	}
int count=0;	
srand((unsigned int)time(NULL));	

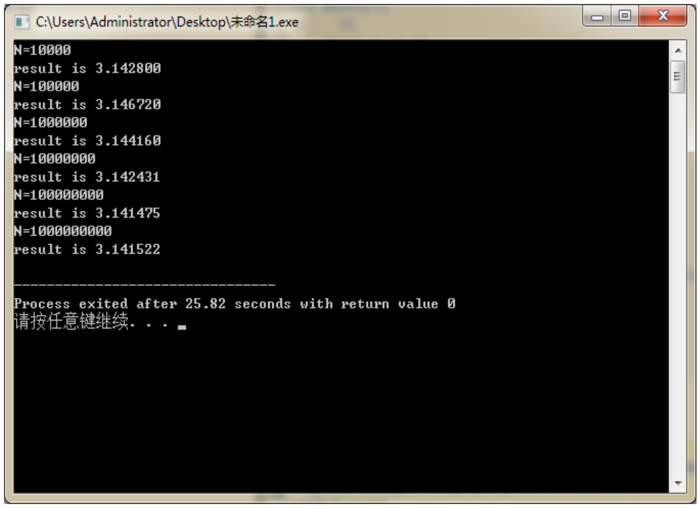


Figure 1: result1

由图可知，不同的n 对应的精确值为：

n	1e4	1e5	1e6	1e7	1e8	1e9
精度	0.01	0.01	0.01	0.01	0.001	0.0001

1.3 设a, b, c 和d 是实数，且a ≤b, c ≤d, f:[a, b] →[c, d] 是一个连续函数，写一概率算法计算积分。

Table 2: Source 2

#include< *stdio.h* >	float y0=f(x);
#include< *stdlib.h* >	if($y > 0 \&\& y < y0$) count++;
#include< *time.h* >	if($y < 0 \&\& y0 < y$) count–;
#define N 1e9	}
float function(float x)	return count/N*(d-c)*(b-a);
{return x*x-1;}	int main()
float call(float(*f)(float),float a,float b,float c,float d)	{
{	srand((unsigned int)time(NULL));
int count=0;	float a=-2,b=2,c=-1,d=3;
for(int i=0;i¡N;i++)	printf("result is %f",call(function,a,b,c,d));
{	return 0;
float x=(float)rand()/RAND_MAX*(b-a)+a;	}
float y=(float)rand()/RAND_MAX*(d-c)+c;	

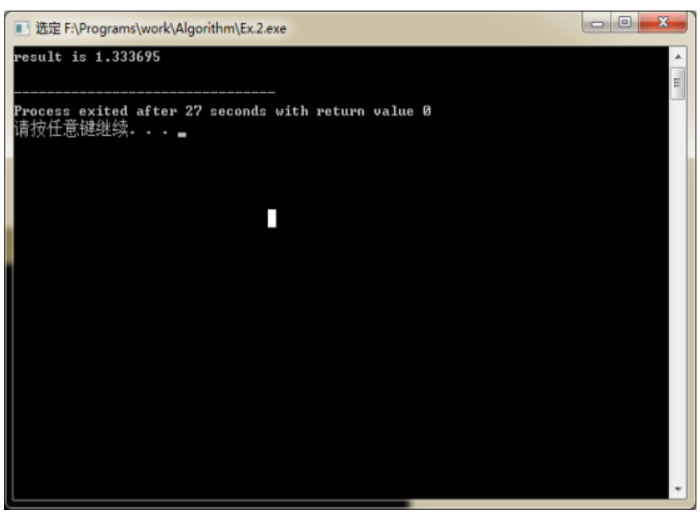


Figure 2: result2

由图可知，实际积分等于4/3。在N=1e9，实验精确度为0.001。

1.4 设ε, σ是(0,1) 之间的常数，证明

若I 是$\int_0^1 f(x)$ 的正确值，h 是由HitorMiss 算法返回的值，则当$n \geq I(1-I)/\varepsilon 2\sigma$ 时有：

$$Prob[|h-I| < \varepsilon] \leq 1-\sigma$$

Proof: 设X 是落入1/4 圆的概率，显然$X \sim (n, I)$, 所以有：

$$E(x) = nI \qquad D(x) = nI(1-I)$$

由切比雪夫不等式可知$P(|nh-nI| < \delta) \geq 1-\frac{D(x)}{\delta^2}$，可得$P(|h-I| < \frac{\delta}{n}) \geq 1-\frac{D(x)}{\delta^2}$，那么令$\frac{\delta}{n} = \varepsilon$, ,$D(x) = nI(1-I)$ 以及带入可得$Prob[|h-I| < \varepsilon] \leq 1-\frac{nI(1-I)}{n^2 * \varepsilon^2}$, 将$n = \frac{I(1-I)}{\varepsilon^2 \sigma}$带入，可得$Prob[|h-I| < \varepsilon] \leq 1-\sigma$

1.5 用上述算法，估计整数子集1 n的大小，并分析n对估计值的影响

Table 3: Source 3

#include< $stdio.h$ >	}
#include< $stdlib.h$ >	int main()
#include< $time.h$ >	{
#include< $string.h$ >	srand((unsigned int)time(NULL));
#define pi 3.1415926	int n;
#define N 10000	for(n=10;n¡=N;n*=10)
int Set[N]={0},visited[N]={0};	{
int setcount(int X_size)	int count=0;
{	for(int j=0;$j < 200$;j++)
for(int temp=0;$temp < N$;temp++)	{
Set[temp]=0,visited[temp]=0;	count+=setcount(n);
int count=0,rand_num=rand()%X_size;	}
do{	printf("n=%d,estimate n is %d",n,count/200);
count++;	}
visited[rand_num]=1;	return 0;
$rand\_num = rand()\%X\_size$;	
}while(visited[rand_num]==0);	
return int((float)(2*count*count)/pi);	

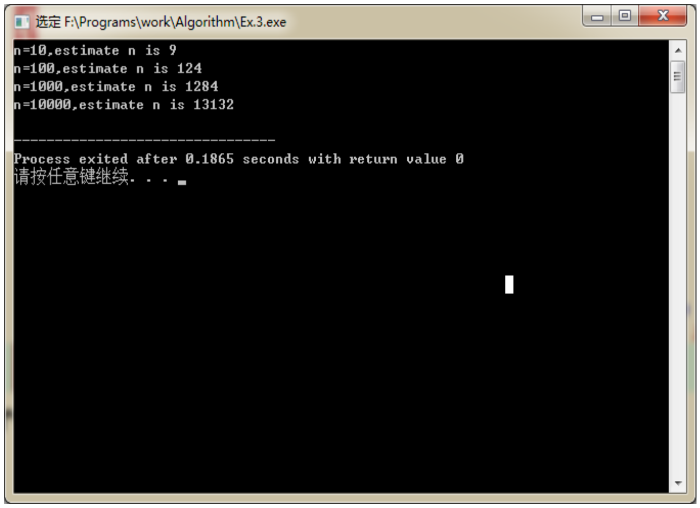


Figure 3: result3

实验采用取200次的平均结果，由结果可知，随着实验的n增大，每次估计都比实际的n 要大，且随着n 增大，误差也在增大，不过效果还可以。

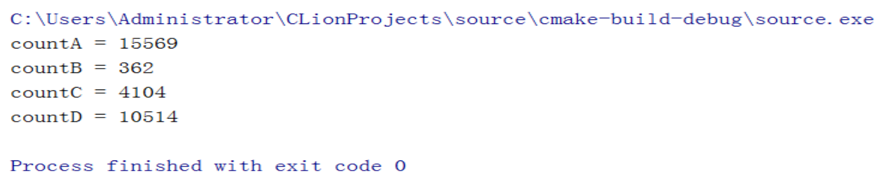
1.6 分析dlogRH的工作原理，指出该算法相应的u和v

dlogRH将原来单独求$\log_{g,p} a$转化为$\log_{g,p} c$其中$c = g^{r+x} mod p$ 相当于做了一个加法变换，其中u变换为$g^{r+x} mod p$,v变化为$(y - r) mod (p - 1)$。

1.7 写一Sherwood算法C，与算法A，B，D比较，给出实验结果

Table 4: Source 4

int Search(int x,int i)	void Gen_Data()
{	{
count =1;	int index, pre;
while($x > val[i]$)	head = (rand()%N+1);
{	val[pre=head] = 1;
i=ptr[i];	for(int i = 2;i¡= N;)
count++}	{index = rand()%N+1 ;
return i;}	if(0==val[index])
int A(int x)	{ val[index] = i;
{ return Search(x,head);}	ptr[pre] = index;
int B(int x)	pre = index;
{	i++;
int y,i=head;	}
int max=val[i];	}
for(int j=1;$j * j \le N$;j++)	int main()
{	{
countB++;	int find_num;
y=val[j];	for(int i = 1; $i <= REAPT\_TIMES$; i++)
if($max < y \&\& y \le x$){i=j;max=y;}	{
}	find_num=rand()%N+1;
return Search(x,i);	A(find_num);
}	$countA+=count$;
int C(int x)	B(find_num);
{	$countB+=count$;
int y,i=head;	C(find_num);
int max=val[i];	$countC+=count$;
for(int j=1;$j < N/8$;j++)	D(find_num);
{	$countD+=count$;
y=val[j];	}
countC++;	printf(”$countA = \%lld \setminus n$”, $countA/REAPT\_TIMES$);
if($max < y \&\& y < x$){i=j;max=y;}	printf(”$countB = \%lld \setminus n$”, $countB/REAPT\_TIMES$);
}	printf(”$countC = \%lld \setminus n$”, $countC/REAPT\_TIMES$);
return Search(x,i);	printf(”$countD = \%lld$”, $countD/REAPT\_TIMES$);
}	return 0;
int D(int x)	
{	
int i=rand()%N+1;y=val[i];	
countD++;	
if($x < y$) {return Search(x,head);}	
if($x > y$){return Search(x,ptr[i]);}	
if($y == x$)return i;	
}	



```
C:\Users\Administrator\CLionProjects\source\cmake-build-debug\source.exe
countA = 15569
countB = 362
countC = 4104
countD = 10514

Process finished with exit code 0
```

Figure 4: result4

实验中同样采用的多次计算取平均值的方法(200),来减少偶然误差，从实验结果来看，使用概率算法查找次数少于其他算法。D算法优于A算法，本算法采用1到$RAND_MAX$(32767)之间随机查找，算法收敛于一半，D 收敛于1/3,而算法B 最优，收敛于$2\sqrt{n}$,而我随机设置大小为n/8,明显低于B算法的$\sqrt{n}$,但也优先于其他算法，理论与实验基本符合。

1.8 当放置（k+1)th皇后时，若有多个位置是开放的,则算法QueensLV 选中其中任一位置的概率相等

证明：满足条件第i个位置别选中的概率为$\frac{1}{i}$，若$j=i$,则意味着在i+1到nb，没有被选中，设选中位置为i的的概率为p(i)。

$$p(i)=\frac{1}{i}*(1-\frac{1}{i+1})*...*(1-\frac{1}{nb})=\frac{1}{nb}$$

1.9 写一算法，求$n = 12$ 20时最优的StepVegas值

Table 5: Source 5

```
bool is_legal(int row, int col)
{
if(row ≥ 2)
for(int m = 1; m < row; m++)
if((col + row) == (chess[m] + m)||(col − row) == (chess[m] − m)||(col == chess[m]))
return false;
return true;
}

bool backtrace(int k)
{
int i = k + 1; int j = 1;
while( i <= CHESS_SIZE&&i >= k + 1 )
{ for(; j ¡= CHESS_SIZE; j++)
if(is_legal(i, j))
{chess[i] = j; st.push(j); i = i + 1; j = 1; break;}
if( j == CHESS_SIZE + 1 )
{ i = i - 1; if(i <= k) return false; j = st.top() + 1;
st.pop(); }
}
if( i <= k) return false;
return true;
}

bool QueenLV()
{
int i, j, nb, k = 0;
if(stepVegas == k) return backtrace(k);
while(true)
{ nb = 0; /* number of open positions for the (k+1)th queen
for(i = 1; i <= CHESS_SSIZE; i + +)
if(is_legal(k+1, i)){nb += 1; if( (rand()%nb + 1) == 1 ) j = i;}
if(nb > 0){k = k + 1; chess[k] = j;}
if(nb == 0||k == stepVegas) break;
}
if(nb > 0) return backtrace(k);
return false;
}
```


算法主要由三部分组成，判断位置是否合法，回退n步，以及QueenLV算法。下

面看实验结果。

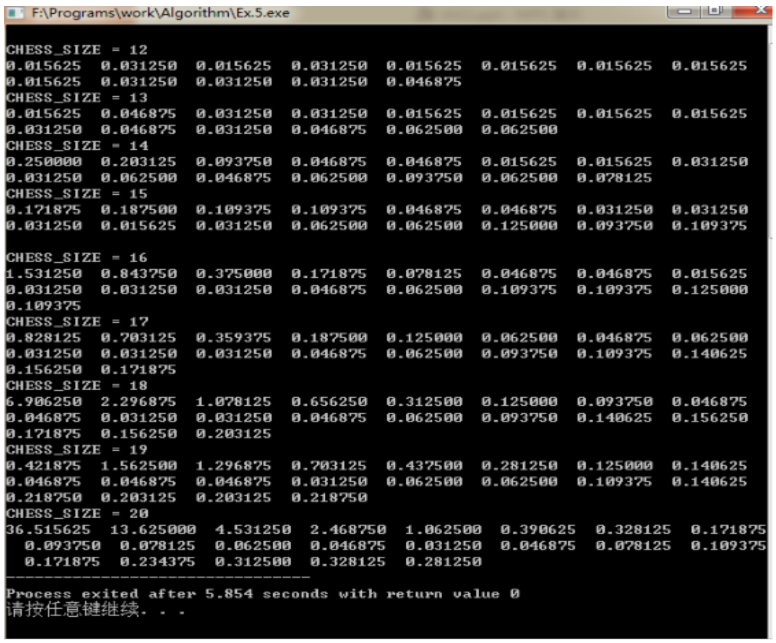


Figure 5: result5

上图是1到20皇后采用不同的stepvagas(1到$CHESE_SIZE$) 所用时间的平均毫秒数（重复200 次），有图可得到下面的结果。

Table 6: 最优的stepVagas对应的步数为

CHESE_SIZE	12	13	14	15	16	17	18	19	20
stepvegas	6	6	7	7	8	10	10	11	12

由实验结果可知，大概在一半的时候，速度最快。特别当CHESE_SIZE变大时，这个趋势会变大。

1.10 与确定性算法相比较，并给出100 10000以内错误的比例

python 代码6：

```
import random
import numpy as np
import math
```

```
def PrintPrimes():
    list=[2,3]
    for n in range(5,10000,2):
        if RepeatMillRab(n,(int)(math.log(n,2))):
            list.append(n)
    return list
def RepeatMillRab(n,k):
    for i in range(1,k+1):
       if MillRob(n)==False:
           return False
    return True
def MillRob(n):
    a=random.randint(2,n-2)
    return Btest(a,n)
def Btest(a,n):
    s=0
    t=n-1
    while(t%2==1):
        s=s+1
        t=t/2
    x=a**t%n
    if (x==1)|(x==n-1):
        return True
    for i in range(1,s):
        x=x*x%n
        if x==n-1:
            return True
    return False
def prime():
    list=[2,3]
    for i in range(5,10000,2):
        flag=0
        for j in range(2,(int)(np.sqrt(i))+1):
            if i%j==0:
                flag=1
        if flag==0:
            list.append(i)
    return list
```

```
if __name__ == "__main__":
    list_certer=prime()
    count=0;
    for i in range(100):
        list_uncerter = PrintPrimes()
        for j in list_uncerter:
            if j not in list_certer:
                count+=1
    print(count)
```

运行了100次，但是只有37个错误，可见错误率非常的低，大约在$37/10/10000 * 100\% = 0.0037\%$,算法的准确非常的高。