数学实验 exp10 实验报告

计 65 赖金霖 2016011377

一、问题背景

对于方阵 $A_{n\times n}$，我们可以定义它的积和式 per(A)如下：

$$per(A)=\sum_{\pi}\prod_{i}^{n}a_{i\pi(i)}$$

其中 π 为 1~n 的排列。本次实验的目的是通过几种方法，计算随机生成的 0/1 矩阵的积和式，并比较各方法的优劣。

1. Naïve 算法

很显然，我们可以枚举 π，直接进行计算，这种算法的时间复杂度为 O(n!)，我们在接下来的内容中称其为 naïve 算法。

2. Ryser 算法

对这个问题，现在已知的最优的确定性算法叫做 Ryser 算法，它把 per(A)变换为

$$per(A)=(-1)^{n}\sum_{S\subseteq\{1,\ldots,n\}}(-1)^{|S|}\prod_{i=1}^{n}\sum_{j\in S}a_{ij}$$

这样，我们只要枚举{1,…,n}的子集，就可以计算出 per(A)了。注意到枚举子集的复杂度为 $O(2^n)$，而枚举子集后的计算是 $O(n^2)$的，所以总时间复杂度为 $O(n^2 2^n)$。当我们使用格雷码枚举子集时，每次枚举的集合 S 和上一次枚举的集合 S’之间只相差一个元素，所以可以设 $b_i=\Sigma_{j\in S}a_{ij}$，每次枚举后 b_i 最多只用加或减一个元素。这样做之后的时间复杂度为 $O(n2^n)$，接下来的 Ryser 算法就是指这一算法。

3. MC 方法

0/1 矩阵的积和式的计算是困难的，可以考虑采用 MC 方法近似。对于一个无偏估计 X， 设 X_1~X_N 与 X 同分布且相互独立，而 $Y=\Sigma^{N}_{i=1}Xi$，那么如果我们需要 Y 是一个以 1-δ 的概率产生一个相对误差不超过ε的估计：

$$P[(1-\epsilon)target\le Y\le(1+\epsilon)target]\ge 1-\delta$$

则需要满足

$$N\ge\frac{E[X^2]}{E[X]^2}\frac{1}{\epsilon^2}log(\frac{1}{\delta})$$

其中 $E[X^2]/E[X]^2$ 被称为 critical ratio。

4. GG 算法

GG 算法是 per(A)的一个无偏估计，它的形式如下

$$GG(A)=det^2(A\odot M_{-1,1})$$

其中⊙为对应项相乘，$M_{-1,1}$ 为和 A 一样大的，各元素以 0.5 概率取-1 或 1 的矩阵。可以证明，GG 算法的 critical ratio 不超过 $3^{n/2}$。通常行列式通过高斯消去计算的复杂度为 $O(n^3)$。

5. KKLLL 算法

KKLLL 算法也是 per(A)的一个无偏估计，它的形式如下

$$KKLLL(A)=\left|det(A\odot M_{-\frac{1}{2}-\frac{\sqrt{3}}{2},-\frac{1}{2}+\frac{\sqrt{3}}{2},1})\right|$$

其中⊙为对应项相乘，$M_{-0.5-0.5\sqrt{3}i,0.5+0.5\sqrt{3}i,1}$为和 A 一样大的，各元素以 1/3 概率取-0.5-0.5√3i-,0.5+0.5√3i 或 1 的矩阵。可以证明，KKLLL 算法的 critical ratio 不超过 $2^{n/2}$。

6. Normal 估计

只要 M 矩阵的各元素是 0 均值 1 方差的随机变量，就能构成一个 per(A)的无偏估计。本次实验中尝试采用 N(0,1)随机产生 M 矩阵，形式如下

$$Normal(A) = det^2(A \odot M_{N(0,1)})$$

二、GG 算法和 KKLLL 算法的理论效率

设 Ryser 算法的时间为

$$C_1 n 2^n$$

GG 算法的时间不超过

$$C_2 n^3 3^{\frac{n}{2}} \frac{log(\frac{1}{\delta})}{\epsilon^2}$$

KKLLL 算法的时间不超过

$$C_3 n^3 2^{\frac{n}{2}} \frac{log(\frac{1}{\delta})}{\epsilon^2}$$

其中 C1、C2、C3 为常数。在具体实现上，我们可以认为 C1、C2 和 C3 是十分接近的数值（彼此的倍数关系很小）。很显然当 n 很小时，Ryser 算法更优，随着 n 增大，GG 算法和 KKLLL 算法效率会变好。

我们可以计算在不同的 δ 和ε下，使得 GG 算法优于 Ryser 算法的理论最小的 n：

δ \ ε	0.2	0.1	0.05	0.01
0.2	88	100	111	136
0.1	91	103	114	139
0.05	94	105	116	141
0.01	97	108	119	144

同理，使得 KKLLL 算法优于 Ryser 算法的理论最小的 n 为：

δ \ ε	0.2	0.1	0.05	0.01
0.2	31	36	40	51
0.1	32	37	42	52
0.05	33	38	43	53
0.01	35	39	44	55

容易看出，在普通的电脑能计算的规模下（如 n≤20 时），GG 算法和 KKLLL 算法没有比 Ryser 算法更优越。当要计算很大的矩阵的积和式时，GG 算法和 KKLLL 算法能优于 Ryser 算法。

三、不同算法的效率对比

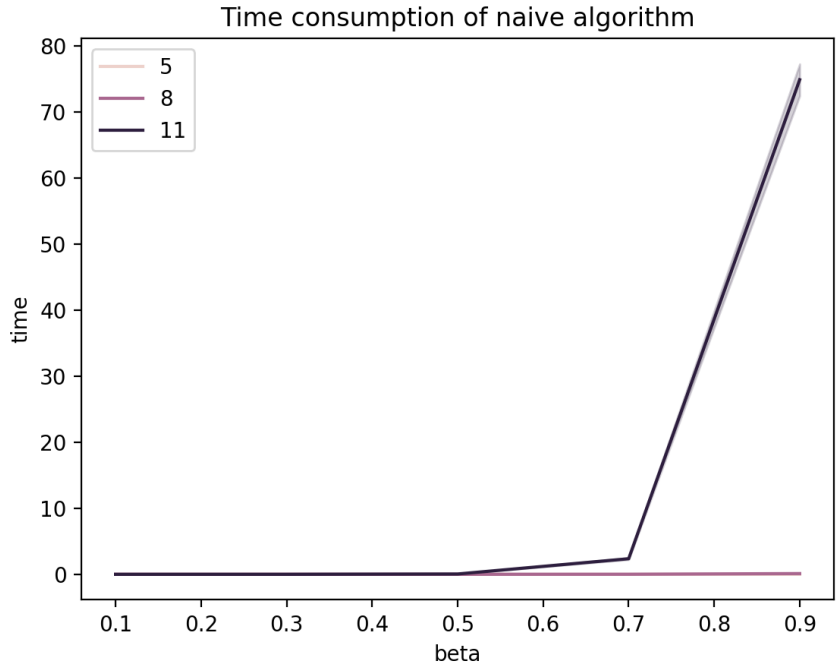
在实验中，我们随机生成矩阵 A，A 中每个元素为 1 的概率记为 β 。

需要注意的是，由于我使用的是不适用于直接科学计算的 Python（且没有像 Python 库函数一样调用 C 语言），所以时间效率要比其他语言如 Matlab 低。

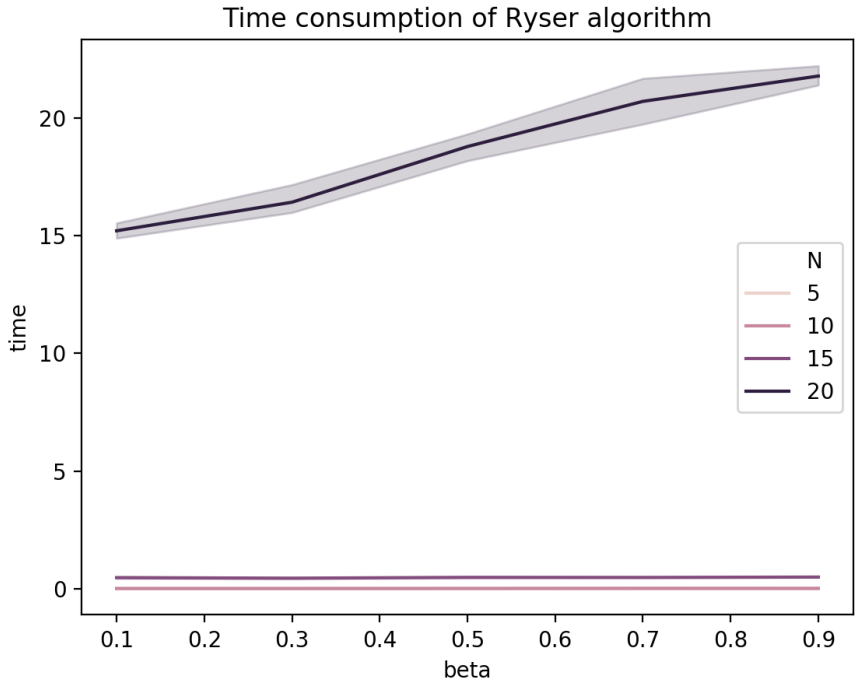
由于 GG 算法、KKLLL 算法和 Normal 估计理论上要求的 N 都很大，在个人电脑上计算不现实，所以对所有 MC 方法，如果不加说明，取 N=20000。

1. 时间效率（本部分图中的 N 指 n，曲线附近 error band 为 ci）

Naïve 算法的时间效率与 β 、n 的关系如下：

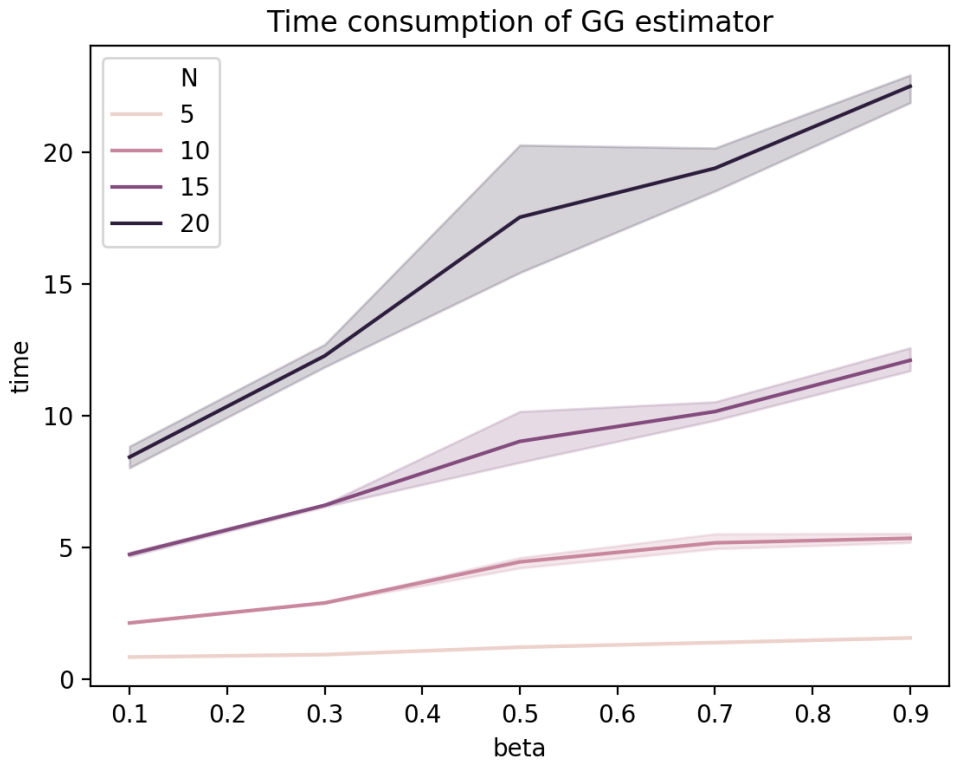


Ryser 算法的时间效率与 β 、n 的关系如下：

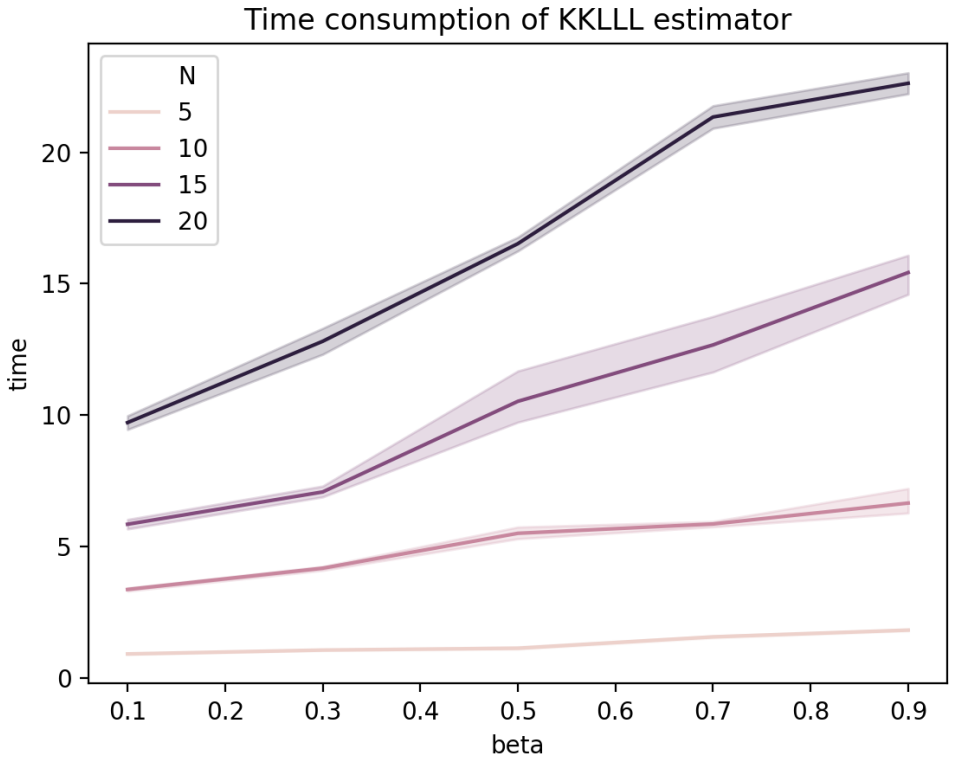


由于 Naïve 算法和 Ryser 算法是指数算法，所以 N 的提升对时间的影响很大。此外，当 n 较大时，可以明显看出算法执行时间随 β 的增大而增大，这可能是因为更多的加减运算导致的。

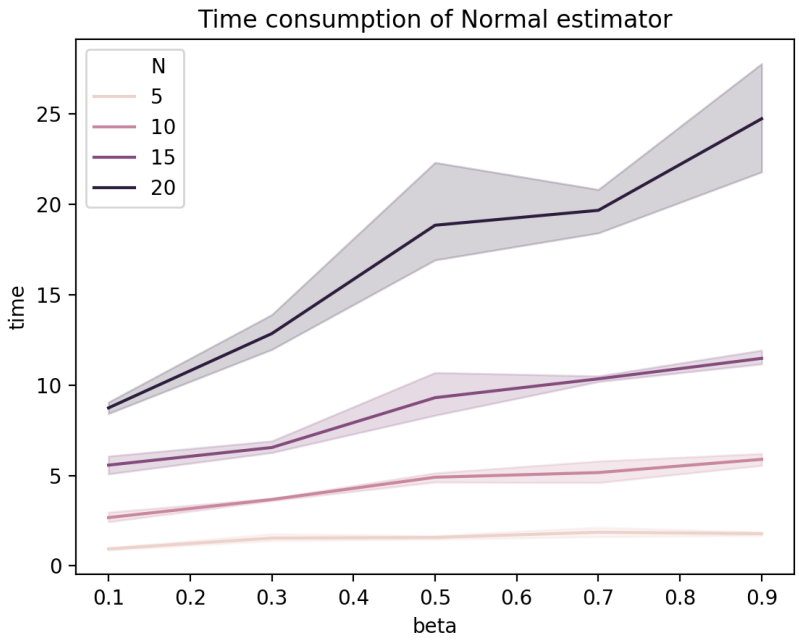
GG 算法的时间效率与 β 、n 的关系如下：



KKLLL 算法的时间效率与 β 、n 的关系如下：



Normal 估计的时间效率与 β 、n 的关系如下：



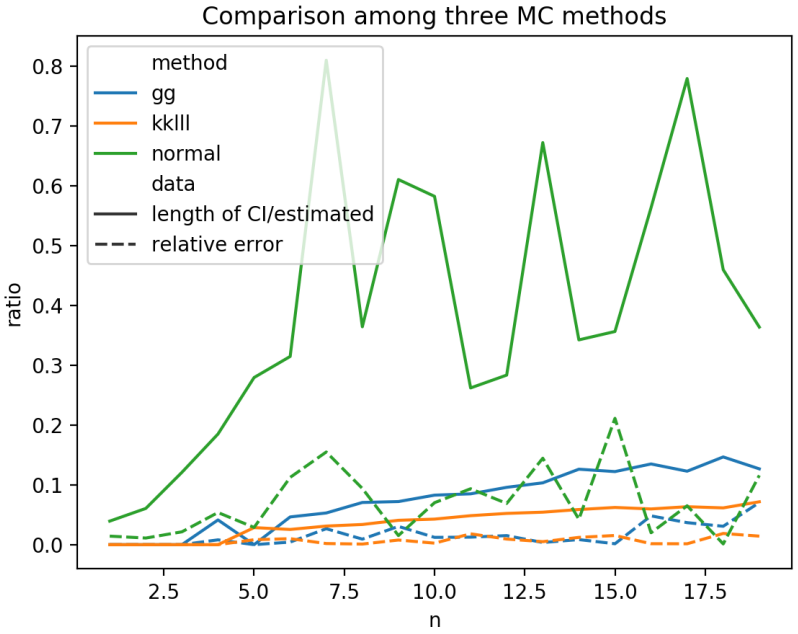
可以看出，MC 方法的运行时间都随 β 和 N 的增大而增大。其中 β 越大，矩阵约稠密，计算行列式的速度越慢；N 越大，矩阵规模越大，行列式计算也越慢。

2. 置信区间与相对误差

为了刻画 MC 方法的结果的精确度，我们引入两个变量“置信区间相对长度 RL”和“相对误差 RE”。设真实值为 real，估计值为 estimated，置信区间长度为 length，则有

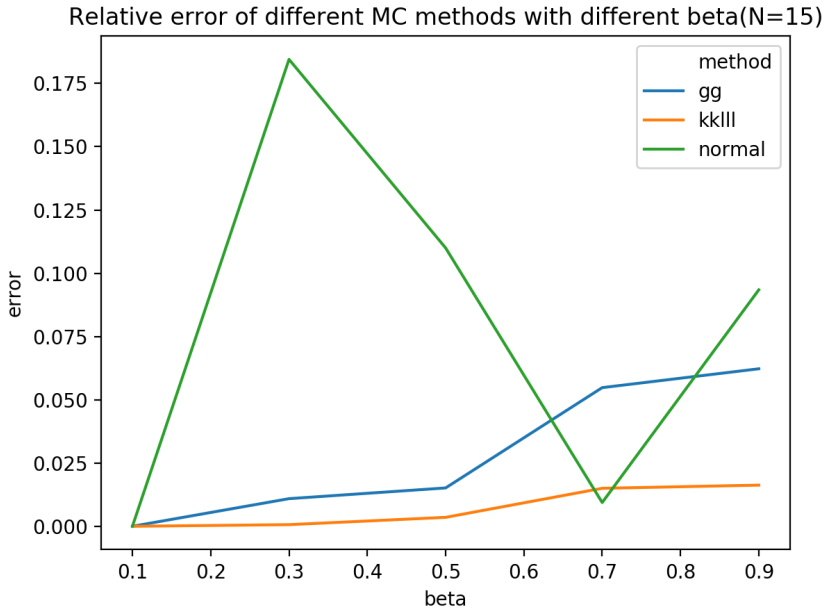
$$RL = \frac{length}{estimated} \qquad RE = \frac{|real - estimated|}{real}$$

对不同的算法，在 β =0.5 时，不同的矩阵规模 n 下，RL 和 RE 如下图：



可以看出，Normal 估计（绿色）的相对置信区间最大，这可能是它不作为实际应用的理由。而 GG 算法（蓝色）和 KKLLL 算法（橙色）中，KKLLL 算法无论是在置信区间上还是相对误差上都比 GG 算法优。此外，随着 n 的增大，各算法的估计偏差也变大，这是因为实验中的样本数 N 是固定的，而为了达到一定的精度而所需的样本数随 n 的增大而增大。

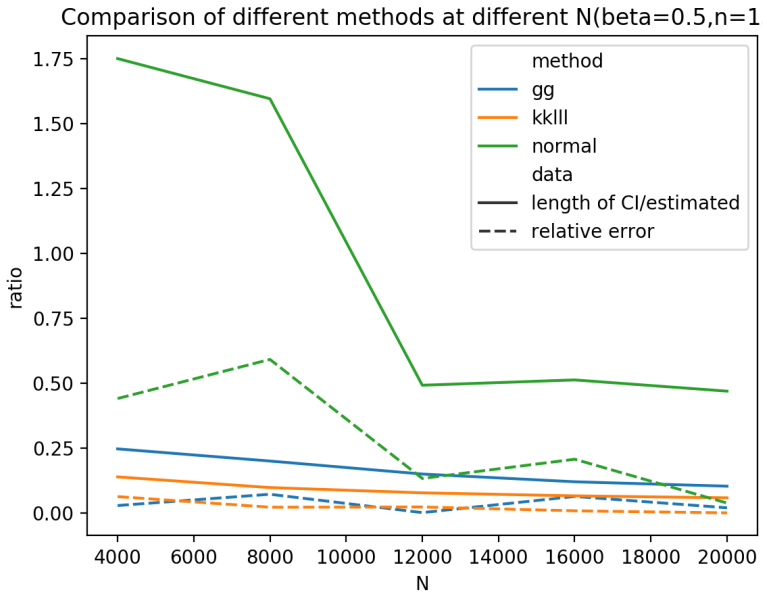
当 N=15 时，各 MC 算法的相对误差随 β 的曲线如下



从总体趋势上看，当 β 越高时，误差越大。

3. MC 方法的 N 的影响

容易知道，对同一个 MC 方法，不同的 N 能产生不同的精度，我们可以对比各 MC 方法在不同的 N 下的 RL 和 RE



在 β 和 n 固定的情况下，对 per(A)估计随 N 的增大而变准。

4. 详细数值

在 n=20， β=0.5，N=20000 时，随机执行一次，各算法的结果为

方法	结果	相对置信区间长度 RL	相对误差 RE
Ryser	4461838126392		
GG	4413282651905	0.1437	0.0109
KKLLL	4581015416762	0.0823	0.0267
Normal	5840527380910	0.8341	0.3090

由于只执行了一次，数值仅供参考。

对不同算法分别执行 10 次，平均时间分别为

方法	Ryser	GG	KKLLL	Normal
时间	17.544s	14.191s	16.561s	16.842s

可以看出，在 MC 方法中，只涉及整数运算的 GG 算法要比复数运算的 KKLLL 算法和实数运算的 Normal 算法更快。

四、代码

代码可以在 https://github.com/lll6924/math_exp/tree/master/exp10 下找到（per.py 和 calculator.py）。

五、参考资料

1. https://en.wikipedia.org/wiki/Computing_the_permanent#Approximate_computation
2. Liang, H. , Shi, L. , Bai, F. , & Liu, X. . (2007). Random path method with pivoting for computing permanents of matrices. Applied Mathematics and Computation, 185(1), 59-71.
3. https://icerm.brown.edu/materials/Slides/sp-s14-w4/Permanent_estimators_via_random_matrices_]_Mark_Rudelson,_Univeristy_of_Michigan.pdf