# 矩阵乘法 实验报告

吴佳龙 2018013418

摘要

本次实验结合理论分析和程序设计，运用了不同算法计算矩阵乘法，具体地，这些方法为：暴力方法，Strassen 算法，改进的 Strassen 算法，并验证了它们的结果正确性，比较了它们的计算时间，实验结果与理论分析相符。并意外地，发现了一个由 CPU 缓存机制影响计算时间的现象。

## 1 问题

比较两个矩阵相乘的常规方法与 Strassen 方法。

说明：为了方便比较，本次实验中限定参与计算的矩阵为方阵，研究方阵规模 $n$ 对于计算效率的影响。

## 2 实验环境

操作系统：Windows 10
IDE：Visual Studio 2017
处理器：3.1 GHz 双核 Intel Core i5

## 3 算法分析

本次实验共两种算法，分别在以下 2 个小节中进行算法描述与分析。

### 3.1 暴力方法

```
void mut_brute(const mat & A, const mat & B, mat & res);
```

**算法描述** 根据矩阵乘法的计算公式

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}, \forall i, j = 1, \cdots, n$$

通过朴素的三重循环实现。

**时间复杂度分析** $\Theta(n^3)$

### 3.2 Strassen 算法及其改进

```
void mut_strassen(const mat & A, const mat & B, mat & res);
void mut_strassen_plus(const mat & A, const mat & B, mat & res);
```

**算法原理** 将 $C = AB$ 分块成 $\frac{n}{2} \times \frac{n}{2}$ 矩阵之间的运算

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

并定义

$$P_1 = a \cdot (f - h), P_2 = (a + b) \cdot h$$
$$P_3 = (c + d) \cdot e, P_4 = d \cdot (g - e)$$
$$P_5 = (a + d) \cdot (e + h), P_6 = (b - d) \cdot (g + h)$$
$$P_7 = (a - c) \cdot (e + f)$$

可以验证有

$$r = P_5 + P_4 - P_2 + P_6$$
$$s = P_1 + P_2$$
$$t = P_3 + P_4$$
$$u = P_5 + P_1 - P_3 - P_7$$

**时间复杂度分析** 每一次分治的过程共需要调用 7 次乘法，这些乘法为规模更小的子问题，其余的矩阵加减操作的复杂度为 $\Theta(n^2)$。因此复杂度满足

$$T(n) = 7T(\frac{n}{2}) + \Theta(n^2)$$

运用主定理解得

$$T(n) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$$

**Strassen 算法的改进** 在运用 Strassen 算法的过程中，发现 Strassen 算法虽然渐进意义下复杂度优于暴力方法，但是在 $n$ 不太大的范围内（本次实验对 $n \leq 2000$ 左右统计了计算时间），实际计算时间远远多于暴力方法。具体结果详见 Section 4。

经过思考，实际实现的 Strassen 算法效率不高的重要原因是：当 $n$ 特别小时，若仍运用 Strassen 算法原理中的计算公式进行计算，复杂度的常数因子极其巨大。

因此提出对于 Strassen 算法的改进：递归调用时，若 $n$ 的大小小于某一阈值 $n_{thre}$，则改为暴力方法计算。这一阈值可以是精心挑选的，理想情况下，它应该是暴力方法与朴素 Strassen 算法的时间曲线的交点。本次实验中，固定阈值 $n_{thre} = 50$，根据实验结果，这已经能大大加快 Strassen 算法的效率，从而超越暴力方法。

**改进的时间复杂度分析** 在渐进意义下，对于 $n \leq n_{thre}$ 的矩阵做乘法，计算复杂度是 $\Theta(1)$ 的。改进的 Strassen 的复杂度为

$$T(n) = \begin{cases} 7T(\frac{n}{2}) + \Theta(n^2), & n > n_{thre} \\ \Theta(1), & n \leq n_{thre} \end{cases}$$

解得

$$T(n) \approx \Theta(1 \cdot (\frac{n}{n_{thre}})^{2.81}) = \Theta(n^{2.81})$$

与朴素 Strassen 算法相同，且具有更小的常数因子。

# 4 结果分析

## 4.1 结果正确性

首先我们验证了算法实现的正确性，验证方法为：随机生成两矩阵 $A, B \in [-100, 100]^{n \times n}$，并调用不同方法计算他们的乘积，再比较不同方法的结果。注意到这里为了防止计算结果超出 int 类型的范围，我们限定矩阵元素位于区间 $[-100, 100]$ 且为整数。

实验表明，对于随机生成的两矩阵，不同方法的计算结果总是相同的。由此可以认为我们对于算法的实现是无误的。

## 4.2 计算时间

对于不同大小的 $n$，统计不同方法进行一次矩阵乘法的计算时间如图 1。

**结果分析** 暴力方法的计算时间随着 $n$ 的增大而呈现一条下凸曲线，这符合 $\Theta(n^3)$ 的时间复杂度。

朴素的 Strassen 算法所需的时间超过了暴力方法，即使它的渐进意义下的复杂度更低，这是由于常数因子过大造成的。另外，在 $n = 300, 400, 500$ 之间，计算时间呈现出阶梯状；在 $n = 600 \sim 1000$ 之间也出现了 90-100 s 左右的阶梯。具体原因由于知识所限无法分析，猜测可能与 CPU 的多级缓存有关。

改进的 Strassen 算法的效率超过了暴力方法，也呈现一条下凸曲线，但增长更加缓慢。

## 4.3 CPU 缓存机制对于计算时间的影响

对一些特殊的 $n$ 调用暴力方法，得到的结果如表 1。

**结果分析** 对于 $n = 512, 1024, 2048$，计算速度都出人意料地慢于它们的邻近值。

结合自己有限的计算机体系结构的相关知识，做出的原因分析如下：在**本次实验的代码实现**中，为了提高效率，将二维矩阵采用一维数组的形式存储，而在某些操作中，需要对矩阵的第一维下标进行枚举，造成了对于内存地址可能的步长为 2 的幂次的连续访问，而这些地址的末几位都是相同的。根据缓存机制，这些地址都被缓存到同一组，不断冲突，带来大量的内存访问，从而使 CPU 缓存的作用大大降低。

因此，CPU 高速缓存友好的代码实现应尽量避免步长为 2 的较大次幂的访问模式，从而避免缓存冲突。

在矩阵乘法这一问题下，解决方式很简单：将规模为 2 的幂次的矩阵存储在稍微大一点

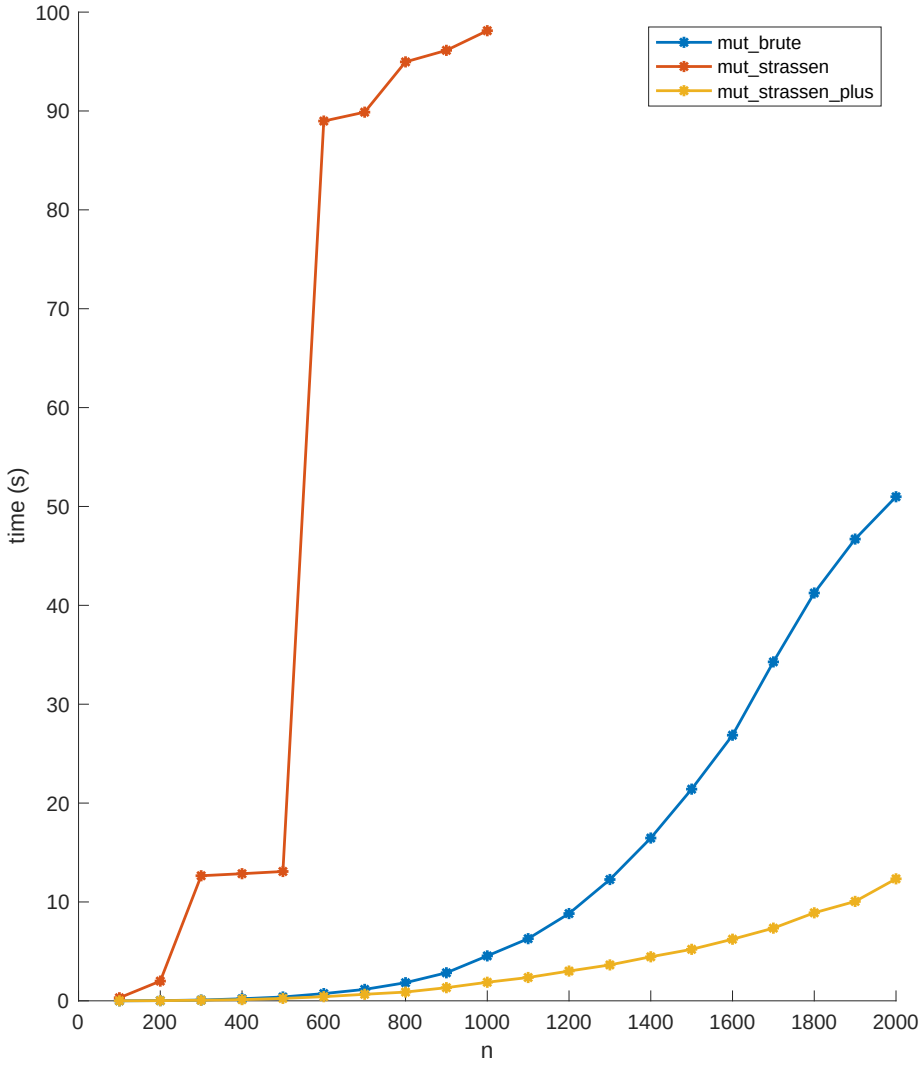


Figure 1: $n$ 在 100 至 2000 之间，不同算法的运行时间。

Table 1: CPU 缓存机制对计算时间的影响

| $n$ | 计算时间 (s) | $n$ | 计算时间 (s) | $n$ | 计算时间 (s) |
|---|---|---|---|---|---|
| 510 | 0.4533672064 | 1022 | 4.9410565644 | 2046 | 62.9813054067 |
| 511 | 0.4090590820 | 1023 | 4.9569864271 | 2047 | 62.7638478806 |
| 512 | **0.6051968601** | 1024 | **16.8471694999** | 2048 | **146.7896244709** |
| 513 | 0.4238428995 | 1025 | 5.0706152361 | 2049 | 62.5015500551 |
| 514 | 0.4265950617 | 1026 | 4.9658465305 | 2050 | 62.9635729621 |

的数组中，比如将 $2048 \times 2048$ 的矩阵存储在 $2049 \times 2049$ 的数组中。

## 5 总结：不同方法的比较

**暴力方法** 实现极其简单，常数因子小，$\Theta(n^3)$ 的复杂度在小范围的 $n$ 也并不比 Strassen 算法慢多少，甚至快于朴素的 Strassen 算法。

**朴素的 Strassen 算法** $\Theta(n^{2.81})$ 的复杂度具有很高的理论价值，但是实现繁琐，且常数因子过大，需要非常精细的实现，且 $n$ 较大的情况下，才能超越暴力方法的运算效率。

**改进的 Strassen 算法** 结合了暴力方法和 Strassen 算法各自的优点，复杂度仍为 $\Theta(n^{2.81})$，但常数因子较小，快于暴力方法。