数据结构第二次上机实验报告

林子开

2023 年 9 月 16 日

目录

1 Ordinary algorithm 的时间复杂度

1.1 理论分析

Ordinary 算法的核心部分如下：

Listing 1: ordinary algorithm 核心部分

```
def ord_matrix_multiplication(A,B):
    """只考虑两个矩阵均为方阵的情况"""
    n = np.size(A,1)
    C = np.zeros((n,n))
    for i in  range(n):
        for j in  range(n):
            for k in  range(n):
                C[i,j] = C[i,j] + A[i,k] * B[k,j]  # 生成C中每个元素需要大约n次乘法和n次加法
    return C
```

为便于讨论，只考虑矩阵 A 和矩阵 B 均为维度大小为 n 的方阵的情况。$C = AB$，在计算 C 中的每个元素 C_{ij} 时，需要计算 n 次乘法和 n 次加法，复杂度为 $\Theta(n)$，由于 C 中一共有 n^2 个元素，则 Ordinary algorithm 算法的时间复杂度为 $\Theta(n^3)$。

1.2 实验结果

分别对矩阵大小 $n = 2^3, 2^4, 2^5, 2^6, 2^7, 2^8$ 的情况进行测试，实验结果如表1所示

表 1: Ordinary Algorithm 的实验结果

矩阵维度 n	2^3	2^4	2^5	2^6	2^7	2^8
time/s	0.000167	0.002	0.016504	0.126088	1.028406	7.999888
$\log_2$(time)	-12.5502	-8.96548	-5.92106	-2.98749	0.04041	2.99998

做出“运行时间-矩阵大小”的双对数图如下

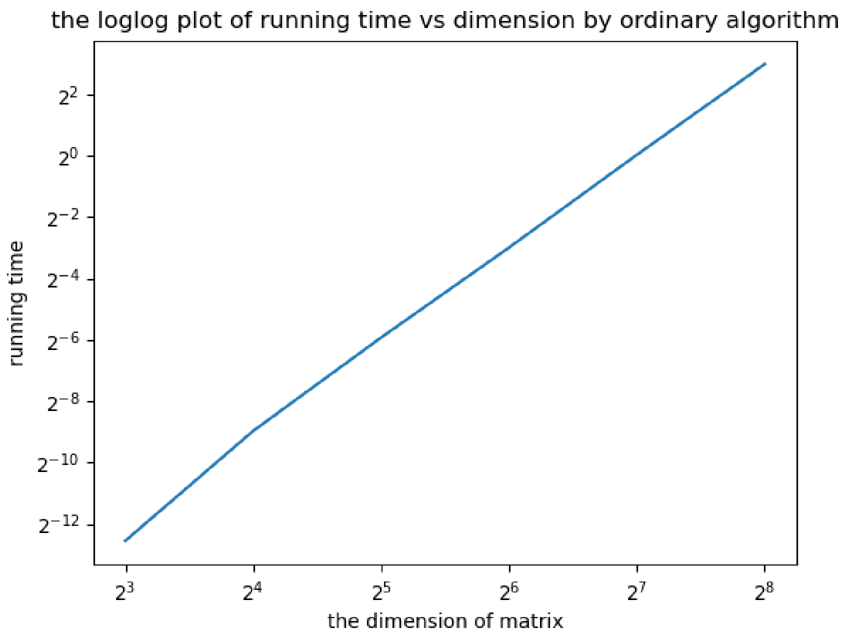

图 1: ordianry algorithm 的时间复杂度双对数

并对 $\log_2$(time) 与 $\log_2(n)$ 用 R 语言做线性回归（即最小二乘法），得到拟合直线的斜率，得到斜率 $k = 3.077$。

1.3 小结

可以看出，实验结果与理论分析较为一致，ordinary algorithm 算法的时间复杂度确实为 $\Theta(n^3)$。

2 Strassen's algorithm 的时间复杂度

2.1 理论分析

为便于讨论，在这一部分，对于 $C = AB$，仍假设 A, B 均为方阵，大小为 2 的整数幂次，以便进行矩阵分块。

Strassen's algorithm 分为以下几个步骤：

1. 递归基础情况，时间复杂度 $\Theta(1)$

Listing 2: Strassen's algorithm 递归基础情况

```
def Strassen_matrix_multiplication(A,B):
    """只考虑方阵，且矩阵维度为2的整数幂次的ing情况"""
    n = np.size(A,1)
    # 若大小为1则直接返回，这是递归的基础情况
    if n == 1:
        return np.multiply(A,B)
```

2. 矩阵分块，时间复杂度 $\Theta(1)$

Listing 3: Strassen's algorithm 矩阵分块

```
    temp =  int(n/2)
    A11 = A[0:temp, 0:temp]
    A12 = A[0:temp, temp:]
    A21 = A[temp:, 0:temp]
    A22 = A[temp:, temp:]
    B11 = A[0:temp, 0:temp]
    B12 = A[0:temp, temp:]
    B21 = A[temp:, 0:temp]
    B22 = A[temp:, temp:]
```

3. 做 10 个矩阵加减法，生成 $S_1 \sim S_{10}$，时间复杂度 $\Theta(n^2)$

Listing 4: Strassen's algorithm 矩阵加减法生成辅助矩阵 S1 到 S10

```
    S1 = B12 - B22
    S2 = A11 + A12
    S3 = A21 + A22
    S4 = B21 - B11
    S5 = A11 + A22
    S6 = B11 + B22
    S7 = A12 - A22
    S8 = B21 + B22
    S9 = A11 - A21
    S10 = B11 + B12
```

4. 做 7 个矩阵乘法，使用递归方法，生成 $P_1 \sim P_7$，时间复杂度 $7T(\frac{n}{2})$

Listing 5: Strassen's algorithm 使用递归方式计算矩阵乘法，生成辅助矩阵 P1 到 P7

```
    P1 = Strassen_matrix_multiplication(A11,S1)
    P2 = Strassen_matrix_multiplication(S2,B22)
    P3 = Strassen_matrix_multiplication(S3,B11)
    P4 = Strassen_matrix_multiplication(A22,S4)
    P5 = Strassen_matrix_multiplication(S5,S6)
    P6 = Strassen_matrix_multiplication(S7,S8)
    P7 = Strassen_matrix_multiplication(S9,S10)
```

5. 做 8 个矩阵加减法，生成分块矩阵 $C_1 \sim C_4$，时间复杂度 $\Theta(n^2)$

Listing 6: Strassen's algorithm 矩阵加减法生成分块矩阵 C1 到 C4

```
    C11 = P5 + P4 - P2 + P6
    C12 = P1 + P2
    C21 = P3 + P4
    C22 = P5 + P1 - P3 - P7
```

6. 分块矩阵拼接，时间复杂度 $\Theta(1)$

Listing 7: Strassen's algorithm 矩阵拼接

```
    C1_ = np.hstack((C11, C12)) # 横向拼接
    C2_ = np.hstack((C21, C22)) # 横向拼接
    return np.vstack((C1_, C2_)) # 垂直拼接
```

由上述讨论可知，Strassen's algorithm 的时间复杂度满足递推公式

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

由于 $n^2 = O(n^{log_2 7-\epsilon})$, $\epsilon = log_2 7 - 2 \approx 0.807 > 0$，根据 Master Method 可知,

$$T(n) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.80735})$$

2.2 实验结果

分别对矩阵大小 $n = 2^3, 2^4, 2^5, 2^6, 2^7, 2^8$ 的情况进行测试，实验结果如表2所示

表 2: Strassen's Algorithm 的实验结果

矩阵维度 n	2^3	2^4	2^5	2^6	2^7	2^8
time/s	0.00183384	0.01380310	0.09567207	0.65380581	4.57863772	31.94628000
$\log_2$(time)	-9.09091947	-6.17886381	-3.38575836	-0.61306589	2.19491842	4.99757604

做出“运行时间-矩阵大小”的双对数图如下

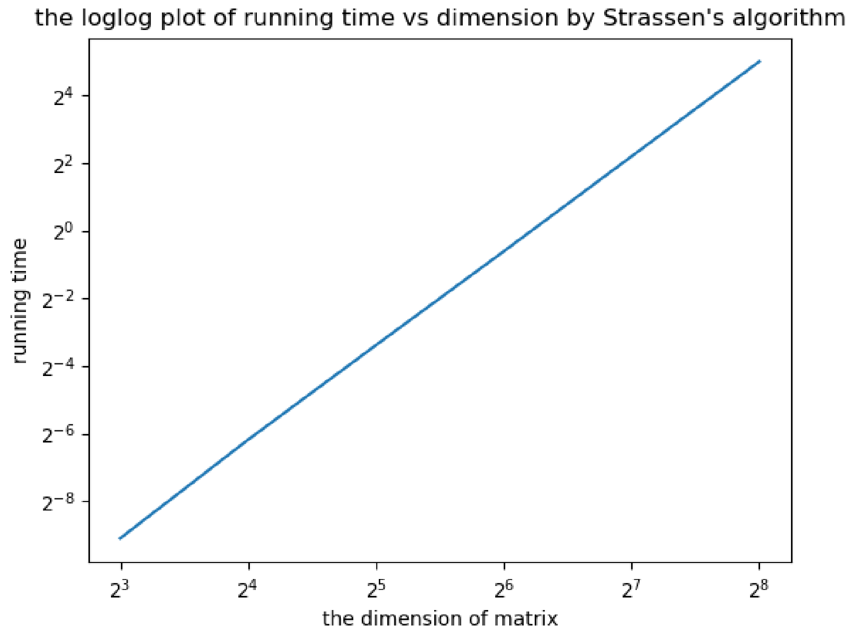

图 2: Strassen’s algorithm 的时间复杂度双对数

并对 $\log_2(\text{time})$ 与 $\log_2(n)$ 用 R 语言做线性回归（即最小二乘法），得到拟合直线的斜率，得到斜率 $k = 2.80962$。

2.3 小结

可以看出，实验结果与理论分析较为一致，Strassen’s algorithm 算法的时间复杂度确实为 $\Theta(n^{\log_2 7})$。

3 附录 A：ordinary algorithm 完整源代码

Listing 8: ordinary algorithm 完整 python 源码

```python
import numpy as np
import time
import matplotlib.pyplot as plt



def ord_matrix_multiplication(A,B):
    """只考虑两个矩阵均为方阵的情况"""
    n = np.size(A,1)
    C = np.zeros((n,n))
    for i in  range(n):
        for j in  range(n):
            for k in  range(n):
                C[i,j] = C[i,j] + A[i,k] * B[k,j]  # 生成C中每个元素需要大约n次乘法和n次加法
    return C

def main():
    time_store = []
```

```python
    start_power = 3
    end_power = 9
    np.random.seed(123)
    for power in  range(start_power,end_power):
        n = 2**power # n为矩阵维度
        A = np.random.randn(n,n)
        B = np.random.randn(n,n)
        start_time = time.time()
        for _ in  range(9-power): # 多次计算求平均值，更准确，但维数太大时，少算几轮
            ord_matrix_multiplication(A,B)
        end_time = time.time()
        print("完成一轮矩阵乘法！",power)
        time_store.append((end_time - start_time)/(9-power))
    print("耗时：", time_store)
    print("log2(耗时）:", np.log2(time_store))
    # fig, ax = plt.subplots(figsize=(8,8))
    plt.loglog(np.exp2( list( range(start_power,end_power))), time_store)
    plt.xscale("log", base=2)
    plt.yscale("log", base=2)
    plt.xlabel("the dimension of matrix")
    plt.ylabel("running time")
    plt.title("the loglog plot of running time vs dimension by ordinary algorithm")
    # plt.show()
    plt.savefig("ord.png")
    return

if __name__=='__main__':
    main()
```

4 附录 B：Strassen's algorithm 完整源代码

Listing 9: Strassen's algorithm 完整 python 源码

```python
import numpy as np
import time
import matplotlib.pyplot as plt

def Strassen_matrix_multiplication(A,B):
    """只考虑方阵，且矩阵维度为2的整数幂次的ing情况"""
    n = np.size(A,1)
    # 若大小为1则直接返回，这是递归的基础情况
    if n == 1:
        return np.multiply(A,B)

    # 首先进行分块
    temp =  int(n/2)
    A11 = A[0:temp, 0:temp]
    A12 = A[0:temp, temp:]
    A21 = A[temp:, 0:temp]
    A22 = A[temp:, temp:]
    B11 = A[0:temp, 0:temp]
```

```python
    B12 = A[0:temp, temp:]
    B21 = A[temp:, 0:temp]
    B22 = A[temp:, temp:]

    # 然后做十个矩阵加法或减法，生成中间的辅助矩阵S1~S10
    S1 = B12 - B22
    S2 = A11 + A12
    S3 = A21 + A22
    S4 = B21 - B11
    S5 = A11 + A22
    S6 = B11 + B22
    S7 = A12 - A22
    S8 = B21 + B22
    S9 = A11 - A21
    S10 = B11 + B12

    # 再做7个矩阵乘法，使用递归算法，生成中间的辅助矩阵P1~P7
    P1 = Strassen_matrix_multiplication(A11,S1)
    P2 = Strassen_matrix_multiplication(S2,B22)
    P3 = Strassen_matrix_multiplication(S3,B11)
    P4 = Strassen_matrix_multiplication(A22,S4)
    P5 = Strassen_matrix_multiplication(S5,S6)
    P6 = Strassen_matrix_multiplication(S7,S8)
    P7 = Strassen_matrix_multiplication(S9,S10)

    # 生成分块矩阵C1~C4
    C11 = P5 + P4 - P2 + P6
    C12 = P1 + P2
    C21 = P3 + P4
    C22 = P5 + P1 - P3 - P7

    C1_ = np.hstack((C11, C12)) # 横向拼接
    C2_ = np.hstack((C21, C22)) # 横向拼接
    return np.vstack((C1_, C2_)) # 垂直拼接


def main():
    time_store = []
    start_power = 3
    end_power = 9
    np.random.seed(123)
    for power in  range(start_power,end_power):
        n = 2**power # n为矩阵维度
        A = np.random.randn(n,n)
        B = np.random.randn(n,n)
        start_time = time.time()
        for _ in  range(9-power): # 多次计算求平均值，更准确，但维数太大时，少算几轮
            Strassen_matrix_multiplication(A,B)
        end_time = time.time()
        print("完成一轮矩阵乘法！",power)
        time_store.append((end_time - start_time)/(9-power))
    print("耗时：", time_store)
```

```
    print("log2(耗时) :", np.log2(time_store))
    # fig, ax = plt.subplots(figsize=(8,8))
    plt.loglog(np.exp2( list( range(start_power,end_power))), time_store)
    plt.xscale("log", base=2)
    plt.yscale("log", base=2)
    plt.xlabel("the dimension of matrix")
    plt.ylabel("running time")
    plt.title("the loglog plot of running time vs dimension by Strassen's algorithm")
    # plt.show()
    plt.savefig("Strassen.png")
    return

if __name__=='__main__':
    main()
```