Data Structure lab2

吴嘉骜 21307130203

2023 年 9 月 15 日

Objective

The objective of this lab is to implement Strassen' s algorithm and ordinary algorithm for matrix multiplication and undertake a comprehensive analysis of both algorithms.

Experiment environment

Windows 11 VsCode Python 3.10.7 64-bit

1 Coding

Write code for Strassen's and ordinary algorithms.

1.1 Ordinary algorithm

Solution: See matrix_mul_ord.py.

```python
# squre matrix multiplication using ordinary algorithm
def matrix_multiply_ordinary(A, B):
n = len(A)
C = [[0 for _ in range(n)] for _ in range(n)]
for i in range(n):
    for j in range(n):
        for k in range(n):
            C[i][j] += A[i][k] * B[k][j]
return C
```

1.2 Strassen's algorithm

Solution: See matrix_mul_Strassen.py.

```python
# squre matrix multiplication using Strassen algorithm, 2^k * 2^k
def matrix_multiply_strassen(A, B):
    n = len(A) # matrix dimension
    if n == 1: # base case
        return [[A[0][0] * B[0][0]]]
```

```
6
7       # divide matrix A and B into four sub-matrices
8       mid = n // 2
9       A11 = [row[:mid] for row in A[:mid]]
10      A12 = [row[mid:] for row in A[:mid]]
11      A21 = [row[:mid] for row in A[mid:]]
12      A22 = [row[mid:] for row in A[mid:]]
13
14      B11 = [row[:mid] for row in B[:mid]]
15      B12 = [row[mid:] for row in B[:mid]]
16      B21 = [row[:mid] for row in B[mid:]]
17      B22 = [row[mid:] for row in B[mid:]]
18
19      # calculate the 7 products of the sub-matrices
20      P1 = matrix_multiply_strassen(A11, matrix_add_sub(B12, B22,2))
21      P2 = matrix_multiply_strassen(matrix_add_sub(A11, A12,1), B22)
22      P3 = matrix_multiply_strassen(matrix_add_sub(A21, A22,1), B11)
23      P4 = matrix_multiply_strassen(A22, matrix_add_sub(B21, B11,2))
24      P5 = matrix_multiply_strassen(matrix_add_sub(A11, A22,1), matrix_add_sub(B11, B22
            ,1))
25      P6 = matrix_multiply_strassen(matrix_add_sub(A12, A22,2), matrix_add_sub(B21, B22
            ,1))
26      P7 = matrix_multiply_strassen(matrix_add_sub(A11, A21,2), matrix_add_sub(B11, B12
            ,1))
27
28      # calculate the four sub-matrices of the result matrix C
29      C11 = matrix_add_sub(matrix_add_sub(matrix_add_sub(P5, P4,1), P2,2), P6,1)
30      C12 = matrix_add_sub(P1, P2,1)
31      C21 = matrix_add_sub(P3, P4,1)
32      C22 = matrix_add_sub(matrix_add_sub(P5, P1,1), matrix_add_sub(P3, P7,1),2)
33
34      # merge the four sub-matrices into one
35      result = []
36      for i in range(n):
37          if i < mid:
38              result.append(C11[i] + C12[i]) # merge C11 and C12
39          else:
40              result.append(C21[i - mid] + C22[i - mid]) # merge C21 and C22
41
42      return result
43
```

```python
def matrix_add_sub(A, B, flag):
    if flag == 1:
        return [[A[i][j] + B[i][j] for j in range(len(A[0]))] for i in range(len(A))]

    if flag == 2:
        return [[A[i][j] - B[i][j] for j in range(len(A[0]))] for i in range(len(A))]
```

2 Analysis

Analyze the theoretical time complexity of two algorithms, evaluate the experimental time overhead associated with the algorithm implementation, and determine their consistency with theoretical computational complexity.

Solution:

2.1 Theoretical time complexity

Ordinary algorithm

The ordinary algorithm is a triple loop, and the innermost loop is a constant time operation, so the time complexity is $O(n^3)$.

Strassen's algorithm

Strassen's algorithm is a recursive algorithm. For every step of recursion, we divide the matrix into 4 sub-matrices with half size, and then calculate the product of the sub-matrices 7 times. Besides, to do matrix addition or subtraction, we need to traverse the matrix, which takes $\Theta(n^2)$ time. The division and combination work takes trivial time. Therefore, the time complexity of the algorithm is $T(n) = 7T(\frac{n}{2}) + \Theta(n^2)$, which is $O(n^{log7})$ according to the master theorem.

Now we use the substitution method to prove the correctness of the time complexity.

Proof:

Assume that $T(k) \leq c_1 k^{log7} - c_2 k^2$ for all $n_0 \leq k < n$ for some constants $c1, c2 > 0$ and n_0. Then consider $n \geq 2n_0$, so that $\frac{n}{2} \geq n_0$ and we can use inductive hypothesis:

$$\begin{aligned} T(n) &= 7T(\frac{n}{2}) + \Theta(n^2) \\ &\leq 7(c_1(\frac{n}{2})^{log7} - c_2(\frac{n}{2})^2) + \Theta(n^2) \\ &= c_1 n^{log7} - c_2 n^2 - \frac{3}{4} c_2 n^2 + \Theta(n^2) \end{aligned}$$

From the definition of big-Θ, we have $\Theta(n^2) \leq \frac{3}{4} c_2 n^2$ for all $n \geq 2n_0$ when c_2 and n_0 are large enough. Therefore, $T(n) \leq c_1 n^{log7} - c_2 n^2$ for all $n \geq 2n_0$.

Now let's consider the base cases $n_0 \leq n < 2n_0$. For every n in this range, the execution time of the algorithm is limited, so as long as c_1 is sufficiently large, we have $T(n) \leq c_1 n^{log7} - c_2 n^2$ for all

$n_0 \leq n < 2n_0$. So the base cases are established.

By the principle of mathematical induction, we have $T(n) \leq c_1 n^{log7} - c_2 n^2 \leq c_1 n^{log7}$ for all $n \geq n_0$. Therefore, the time complexity of the algorithm is $O(n^{log7})$.

2.2 Experimental time overhead

We use the following code to test the time overhead of the two algorithms. See timecost.py.

```
from matrix_mul_ord import matrix_multiply_ordinary
# from matrix_mul_Strassen import matrix_multiply_strassen
from matrix_mul_Strassen_re import matrix_multiply_strassen_t
import time
import matplotlib.pyplot as plt

time_ord = []
time_strassen = []
time_strassen_16 = []
time_strassen_32 = []
time_strassen_64 = []
nrange = []
for k in range(2,11):
    nrange.append(2**k)
for n in nrange:
    # construct matrix A and B
    A = [[1 for _ in range(n)] for _ in range(n)]
    B = [[1 for _ in range(n)] for _ in range(n)]
    start = time.time()
    result_ord = matrix_multiply_ordinary(A, B)
    end = time.time()
    time_ord.append(end - start)

    # start = time.time()
    # result_strassen = matrix_multiply_strassen(A, B)
    # end = time.time()
    # time_strassen.append(end - start)

    start = time.time()
    result_strassen_16 = matrix_multiply_strassen_t(A, B, 16)
    end = time.time()
    time_strassen_16.append(end - start)

    start = time.time()
```

```
    result_strassen_32 = matrix_multiply_strassen_t(A, B, 32)
    end = time.time()
    time_strassen_32.append(end - start)

    start = time.time()
    result_strassen_64 = matrix_multiply_strassen_t(A, B, 64)
    end = time.time()
    time_strassen_64.append(end - start)

# print the time cost
print('Ordinary Multiplication:',time_ord)
print('Strassen16 Multiplication:',time_strassen_16)
print('Strassen32 Multiplication:',time_strassen_32)
print('Strassen64 Multiplication:',time_strassen_64)

# plot the time cost
plt.figure(figsize=(10, 6))
plt.plot(nrange, time_ord, label='Ordinary Multiplication')
# plt.plot(nrange, time_strassen, label='Strassen Multiplication')
plt.plot(nrange, time_strassen_16, label='Strassen16 Multiplication')
plt.plot(nrange, time_strassen_32, label='Strassen32 Multiplication')
plt.plot(nrange, time_strassen_64, label='Strassen64 Multiplication')
plt.xlabel('Matrix Size (N)')
plt.ylabel('Time (seconds)')
plt.title('Matrix Multiplication Algorithms Comparison')
plt.legend()
plt.grid(True)
plt.show()
```

Code interpretation and result analysis:

The code is used to test the respective time cost of the two algorithms. We generate all-one matrices whose sizes range from 2^2 to 2^8, and record each running time. We plot the time cost of the two algorithms in Figure 1.

From the figure, we can see that the time cost of Strassen's algorithm increases rapidly with the increase of the matrix size, far exceeding the ordinary algorithm. This is beyond my expectation. I tried the following methods to tackle this situation.

First, I thought that the matrix size is not large enough, and Strassen's algorithm is slower than the ordinary because of the recursions with more additions and subtractions, and larger constant factor. Then I tried to increase the matrix size to 2^{10}, but it takes too much time to run, and I also recalled that the slides said that Strassen's algorithm beats the ordinary algorithm on today's machines for $n \geq 32$ or so. So it may not be a good idea to increase the matrix size.

After finding some information on the Internet, I found that my result is consistent with the actual situation. The reason is that using Strassen's algorithm for recursive operations requires creating a large number of dynamic two-dimensional arrays, and allocating heap memory space will consume a lot of computation time, thereby masking the advantages of the Strassen algorithm. Therefore, I tried to improve the algorithm by setting a threshold t. When n is less than t, the matrix is calculated using the ordinary method instead of continuing with divide and conquer recursion. I set the threshold to $16, 32, 64$ respectively in the revised code matrix_mul_Strassen_re.py, and the new running results are shown in Table 1 and Figure 2.

```python
# squre matrix multiplication using revised Strassen algorithm, 2^k * 2^k
def matrix_multiply_strassen_t(A, B, t):
    n = len(A)
    if n <= t: # set the threshold
        C = [[0 for _ in range(n)] for _ in range(n)]
        for i in range(n):
            for j in range(n):
                for k in range(n):
                    C[i][j] += A[i][k] * B[k][j]
        return C

    # divide matrix A and B into four sub-matrices
    mid = n // 2
    ...
    # the rest is the same as the original Strassen's algorithm
```

Table 1: Running time of different matrix multiplication algorithms.

Matrix Dimension	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
Ordinary Multiplication (ms)	0.0	0.0	1.0	3.0	21.9	186.5	1593.8	14359.1	135279.0
Strassen16 Multiplication (ms)	0.0	0.0	1.0	2.0	24.9	165.8	1229.5	8931.1	80677.6
Strassen32 Multiplication (ms)	0.0	0.0	1.0	3.0	24.0	161.6	1165.1	8708.5	74905.4
Strassen64 Multiplication (ms)	0.0	0.0	1.0	3.0	23.9	163.6	1247.9	9895.1	123558.6

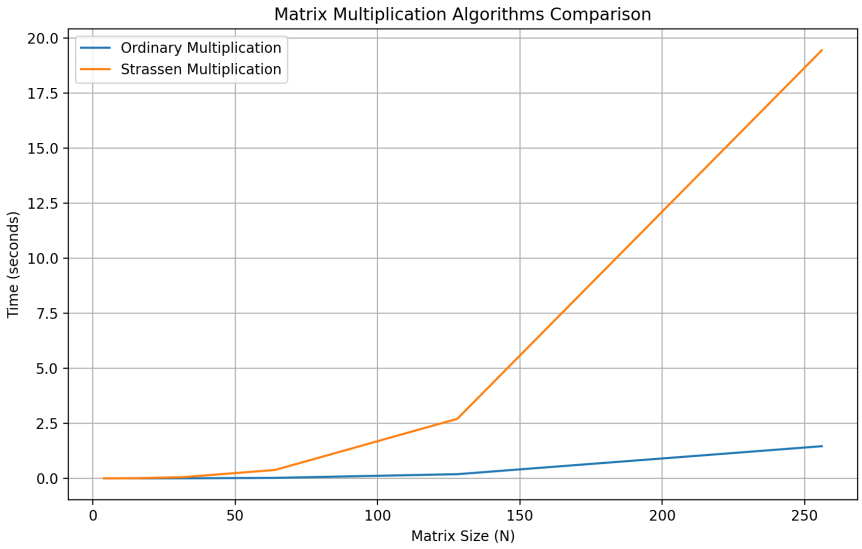

Figure 1: Time cost of the ordinary algorithm and Strassen's algorithm

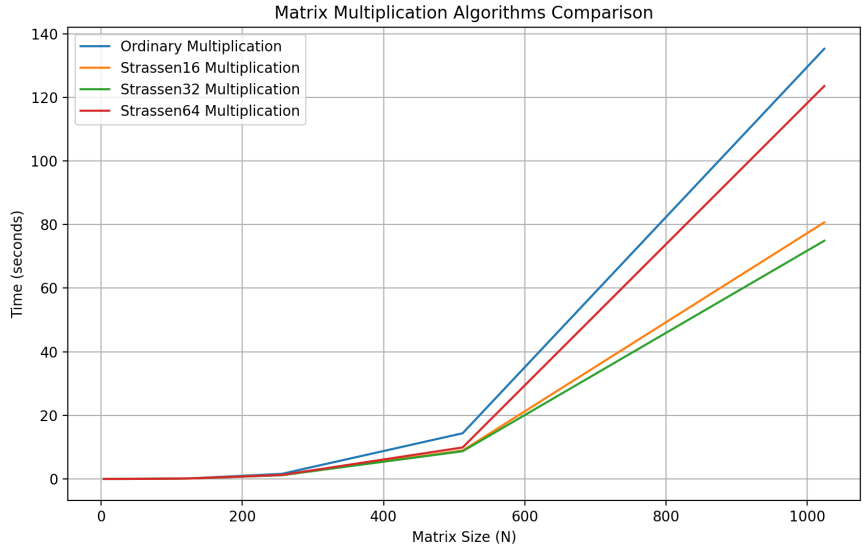

Figure 2: Time cost of the ordinary algorithm and Strassen's algorithm with threshold

After improvement, the advantage of Strassen's algorithm is obvious when the matrix size is large, and the computation time is greatly reduced than the ordinary algorithm. This is consistent with the theoretical time complexity analysis.

Apart from that, we can see that $t = 32$ is good with the lowest cost among three thresholds. To further verify that the orders of magnitude of the two algorithms are correct, we take the cube root of the running time of the ordinary algorithm, take $log7$ root of the running time of Strassen's algorithm, and draw a scatter plot. We use $scikit - learn$ package in Python to calculate both the linear regression models

and compute their correlation coefficients to varify the linear relation. The closer the absolute value of coefficient is to 1, the stronger the linear correlation is. Finally, we plot the regression lines. See code in timeana.py which is omitted here and results are shown in Table 2 and Figure 3 below.

Table 2: Regression and Correlation Results

	Ordinary Method	**Strassen32 Method**
Regression Equation	$y = 0.0050x - 0.0426$	$y = 0.0045x - 0.0399$
Correlation Coefficient	0.9995	0.9993

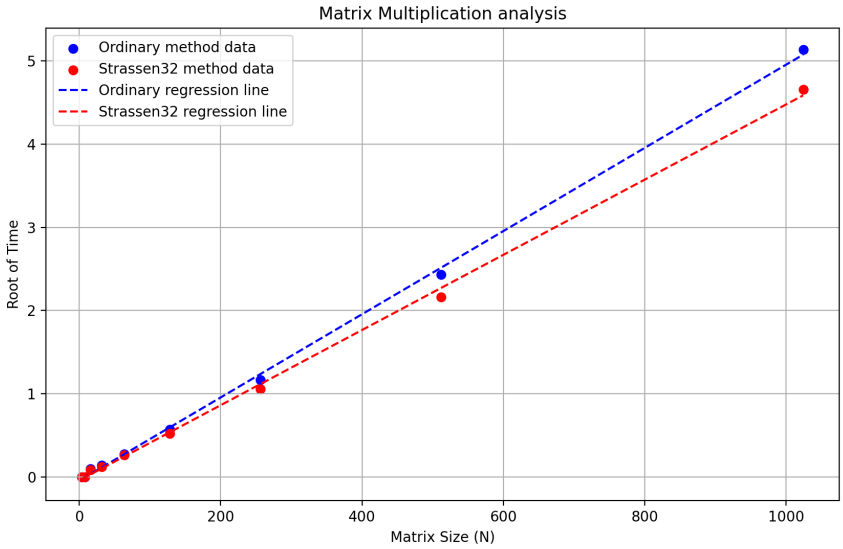

Figure 3: Rooted time cost of the ordinary algorithm and Strassen's algorithm with threshold

From the table and the figure, we can see that the rooted time cost of the algorithm is much closed to a linear function of the matrix size, which is consistent with the theoretical time complexity analysis.

Conclusion:

The theoretical time complexity of the ordinary algorithm is $O(n^3)$, and the theoretical time complexity of Strassen's algorithm is $O(n^{log^7})$. When the matrix size is large enough, Strassen's algorithm with thresholds is much faster than the ordinary algorithm. However, when the matrix size is small, the ordinary algorithm is still useful.