FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION OF HIGHER EDUCATION ITMO UNIVERSITY

Report
on the practical task No. 8
"Practical analysis of advanced algorithms"

Performed by

Rami Al-Naim

J4132c

Accepted by

Dr Petr Chunaev

St. Petersburg
2020

1 Goal

Practical analysis of advanced algorithms.

2 Formulation of the problem

During this work two algorithms from "Thomas H. Cormen Charles E. Leiserson Ronald L. Rivest Clifford Stein Introduction to Algorithms" book were chosen and analyzed.

For this work the following algorithms were chosen: Matrix-chain multiplication using dynamic programming and Strassen method for matrix multiplication.

3 Brief theoretical part

Matrix multiplication is a very time consuming operation with naive implementation having a $O(n^3)$ time complexity. In case of matrix-chain multiplication (1) the number of scalar multiplications influence the runtime the most. The multiplication of two matrix $A_{p\times k}$ and $B_{k\times q}$ requires pkq multiplications.

$$A_1A_2A_3\ldots A_n \tag{1}$$

The goal of the algorithm is to find the optimal parenthesization of the product with the least amount of scalar multiplications. For example, three matrices $A_{1,\ 100\times20}, A_{2,\ 20\times30}, A_{3,\ 30\times10}$ could be multiplied in the following order:

- $((A_1A_2)A_3) \longrightarrow 100\cdot20\cdot30 + 100\cdot30\cdot10 \longrightarrow 90000$ scalar multiplications;

- $(A_1(A_2A_3)) \longrightarrow 20\cdot30\cdot10 + 100\cdot20\cdot10 \longrightarrow 26000$ scalar multiplications.

Algorithm finds the optimum by dividing the equation (1) into subequations and searching for their optimums since the final optimum is the combination of optimum parenthesization of the subequations. The algorithm iteratively finds the best parenthesization for all the subequations and forms the table (or 2D array) which can be used to define the order of multiplication. The algorithm works for $O(n^3)$ time where n is the number of matrices. Despite that it can save time during the multiplication of large matrices. It uses $\Theta(n^2)$ space to store matrix of optimal number of scalar multiplications m and the matrix of optimal parenthesization s.

Strassen method for matrix multiplication allows to multiply matrix with time complexity of $O(n^{2.81})$ instead of $O(n^3)$. This "Divide-and-Conquer" algorithm divides $n\times n$ matrices into four matrices $n/2\times n/2$ matrices and multiple then using novice or recursive algorithm in a specific manner. Recurrent matrix multiplication takes $O(n^3)$ time, and Strassen method reduces overall amount of matrix multiplication at each step by one. This allows to reduce time compleity to $O(n^{2.81})$ or $O(n^{ln7})$.

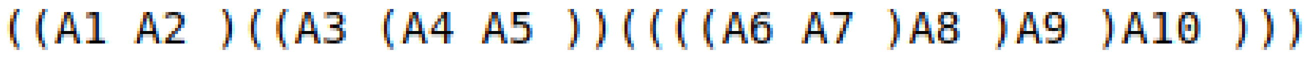
This work was done on `Python3.8`. Due to the architecture and internal principals of this language, recursion works much longer than iterative algorithms. Since that, the recursive Strassen method was implemented, however, due to constrains of the hardware in my disposal the experiments were performed on Strassen algorithm where recursive multiplication was substituted with naive approach. Such algorithm will have time complexity of $O(n^3)$, but still should perform better than fully-naive approach.

4 Results

4.1 Matrix-chain multiplication

In order to analyze the algorithm, the novice approach for matrix multiplication was used. The `numpy` implementation of matrix multiplication uses precompiled C code and this can greatly outperform many other approaches.

The following experiment was conducted: the dimensions of the matrix were randomly generated from the interval of $[10, 100]$, and random matrices with random numbers from the interval $[0, 1]$ and those dimensions are generated. The aforementioned algorithm is used to find the optimal parenthesization. Then, the elapsed time on the algorithm and optimal multiplication compared to straightforward novice multiplication. Moreover, the number of scalar multiplications is also compared. The example of the optimal parenthesization for chain of 10 matrices is present on the Figure 1.



```
((A1 A2 )((A3 (A4 A5 ))((((A6 A7 )A8 )A9 )A10 )))
```

Figure 1: Example of optimal parenthesization for chain of 10 matrices

The results of comparisons are present in the Table 1.

# of matrices	5		10		50		100	
Algorithm	DP	N	DP	N	DP	N	DP	N
n, 10^3	356	925	446	454	1670	5914	3395	24268
t, ms	127	182	294	985	1381	2350	3200	20000

Table 1: Result of the experiment. DP - dynamic programming approach, N - novice solution, t - elapsed time in ms, n - number of scalar multiplications. In case of DP t includes the multiplication time and optimal parenthesization search.

4.2 Strassen algorithm

The experiment was conducted in the following manner: each algorithm was run 10 times for the same matrices with dimensions of power of two up to 9th. The average time was computed for each dimension of the input matrices.

The result of comparison are present in Table 2.

N	t_n, ms	t_s, ms
1	0.02	0.31
2	0.04	1.92
4	0.21	2.21
8	1.53	2.56
16	6.66	8.99
32	56.09	55.78
64	510.94	423.18
128	4203.01	3999.32
256	24901.37	22416.71
512	203164.7	172872.95

Table 2: Result of the experiment, t_n - runtime of the novice algorithm, t_s - runtime of the Strassen algorithm.

5 Conclusions

During this work two algorithms were implemented and analyzed.

The Dynamic Programming approach was used for finding the optimal parenthesization for matrix-chain multiplication. As it can be seen from the Table 1, despite the $O(n^2)$ time complexity, in case of large numbers of matrix it allows to greatly speedup the computations without any accuracy losses.

The Strassen algorithm was implemented and compared with naive matrix multiplication. The result of the experiment is present in Table 2. It yields better result compared to naive "3-nested-loop" algorithm when the dimension of the matrices reaches 32. It can be explained with a fact that while Strassen algorithm operates smaller matrices and decreases the amount of matrix multiplications, it requests more function calls. Function calls considered to have constant time complexity, but on a small matrices those calls could impact the algorithm's runtime.

Appendix

Project file available here: GitHub.