CS 412: Algorithms 2
Project Report
Analyzing Matrix Multiplication

Sandesh Kumar sk05567@alumni.habib.edu.pk,
Neha Valliani nv06229@alumni.habib.edu.pk,
Muhammad Bilal Fayaz mf05942@alumni.habib.edu.pk,
Hammad Ahmed ha05890@alumni.habib.edu.pk

30th April 2022

1 Introduction

Our project focuses on the seemingly trivial task of matrix multiplication. The twist springs up at optimizing the number of matrices being multiplied. Since matrix multiplication is not commutative, we are relieved of some problem-solving. However, due to the associative nature of matrix multiplication, we are left to find the best parenthesising of the matrices; so a minimum number of scalar multiplications has to be done. These multiplications can reach up to a very large number and similarly can be reduced to the most efficient multiplicative task. Our project is divided into 2 parts. Firstly, matrix multiplication between 2 matrices. And then matrix multiplication between 'n' matrices. In both cases, we have to optimize the multiplication in a way that the number of constant multiplications are reduced. This problem of finding the optimal solution can be tackled in several ways. In our report, we are going to analyze a few design techniques we have learnt through this course, in addition to that some traditional algorithms like the Strassen's algorithm and the more popular brute force method will be under our digital scope. We aim to analyze these methods and design choices both empirically and theoretically and share the results annotated with our understanding.

2 Algorithms

The algorithms and design techniques that we plan to implement are:
Naive method (Brute force) for matrix multiplication.
Naive method (Brute force) for chain matrix multiplication.
Strassen's for matrix multiplication.
Dynamic Programming for chain matrix multiplication.

3 Repository

Our repository with the all related documents and the codes used in solving the mentioned problems and analyzing them.

4 Implementation

4.1 For Square Matrix Multiplication

```python
mat_A = np.random.randint(20, size=(128,128))
mat_B = np.random.randint (20, size=(128,128))
n = 128

mat_C = [[0 for y in range (n)] for x in range (n)]

for i in range (n):
    for j in range (n):
        mat_C[i][j] = 0
        for k in range (n):
            mat_C[i][j] += mat_A[i][k] * mat_B[k][j]

print(mat_C)
```

Code Listing 1: Brute Force for matrix multiplication

```python
import numpy as np

mat_A = np.random.randint(20, size=(128,128))
print(mat_A,"mata", len(mat_A))
mat_B = np.random.randint (20, size=(128,128))
print(mat_B,'matb', len(mat_B))
def split(mat):
    r, c = mat.shape
    r_1, c_1 = r//2, c//2
    return mat[:r_1, :c_1], mat[:r_1, c_1:], mat[r_1:, :c_1], mat[r_1:, c_1:]

def strassen(x, y):
    if x.shape[0] == 1:
        return x * y

    else:
        mat_A_q1, mat_A_q2, mat_A_q3, mat_A_q4 = split(x)
        mat_B_q1, mat_B_q2, mat_B_q3, mat_B_q4 = split(y)

        prod_1 = strassen(mat_A_q1, mat_B_q2 - mat_B_q4)
        prod_2 = strassen(mat_A_q1 + mat_A_q2, mat_B_q4)
        prod_3 = strassen(mat_A_q3 + mat_A_q4, mat_B_q1)
        prod_4 = strassen(mat_A_q4, mat_B_q3 - mat_B_q1)
        prod_5 = strassen(mat_A_q1 + mat_A_q4, mat_B_q1 + mat_B_q4)
        prod_6 = strassen(mat_A_q2 - mat_A_q4, mat_B_q3 + mat_B_q4)
        prod_7 = strassen(mat_A_q1 - mat_A_q3, mat_B_q1 + mat_B_q2)

        mat_c_q1 = prod_5 + prod_4 - prod_2 + prod_6
        mat_c_q2 = prod_1 + prod_2
        mat_c_q3 = prod_3 + prod_4
        mat_c_q4 = prod_1 + prod_5 - prod_3 - prod_7

        mat_C = np.vstack((np.hstack((mat_c_q1, mat_c_q2)), np.hstack((mat_c_q3,
    mat_c_q4))))
        return mat_C
x=mat_A
y=mat_B

result = strassen(mat_A, mat_B)
print(result)
print(len(result))


```

Code Listing 2: Strassen's method for matrix multiplication

4.2 For Chain Matrix Multiplication

```
import numpy as np
import random as rd

def bruteForceChainMatrix():
    x=rd.randint(1,20)
    n= rd.randint(1,20)
    Papa_mat = []
    n_steps=[x]
    total=0
    #genrating random numbers
    for i in range(n):
        y=rd.randint(1,20)
        Matrix = np.random.randint(20, size=(x,y))
        print(x,y)
        Papa_mat.append(Matrix)
        n_steps.append(x)
        x=y

    for i in range(2,len(n_steps)):
        total+=n_steps[i]*n_steps[i-1]*n_steps[0]
    print(total) #The total number of steps required to multiply the matrices



    #making the first matrix
    mat_A = Papa_mat[0]
    mat_B = Papa_mat[1]
    # n_steps = len(mat_A[0]) * len(mat_A[1]) * len(mat_B[0])
    mat_C = np.zeros((len(mat_A[1]), len(mat_B[0])))
    for i in range (len(mat_A[0])):
        for j in range (len(mat_B[1])):
            mat_C[i][j] = 0
            for k in range (len(mat_A[1])):
                mat_C[i][j] += mat_A[i][k] * mat_B[k][j]
    #doing further multiplications
    for f in  range(2,len(Papa_mat)):
        miniPapa = Papa_mat[f]
        temp = np.zeros((len(mat_C[1]), len(miniPapa[0])))
        # n_steps = n_steps + (len(mat_A[0]) * len(mat_A[1]) * len(mat_B[0]))
```

```python
        # mat_C = np.matmul(mat_C, Papa_mat[f])
        for final1 in range(len(mat_C[0])):
            for final2 in range(len(miniPapa[1])):
                for final3 in range(len(mat_C[1])):
                    temp[final1][final2] += mat_C[final1][final3] * miniPapa[final3
    ][final2]
                    mat_C = temp
    return mat_C

print(bruteForceChainMatrix())


```

Code Listing 3: Brute Force for Chain Matrix Multiplication

```python
def MatrixChainOrder(p, n):
    m = [[0 for x in range(n)] for x in range(n)]
    for i in range(1, n):
        m[i][i] = 0
    for L in range(2, n):
        for i in range(1, n-L + 1):
            j = i + L-1
            m[i][j] = float("inf")
            for k in range(i, j):
                q = m[i][k] + m[k + 1][j] + p[i-1]*p[k]*p[j]
                if q < m[i][j]:
                    m[i][j] = q

    return m[1][n-1]
```

Code Listing 4: Dynamic Programming for Optimized Chain Matrix Multiplication

5 Theoretical Analysis And Further Explorations

5.1 Brute force for Square Matrix multiplication

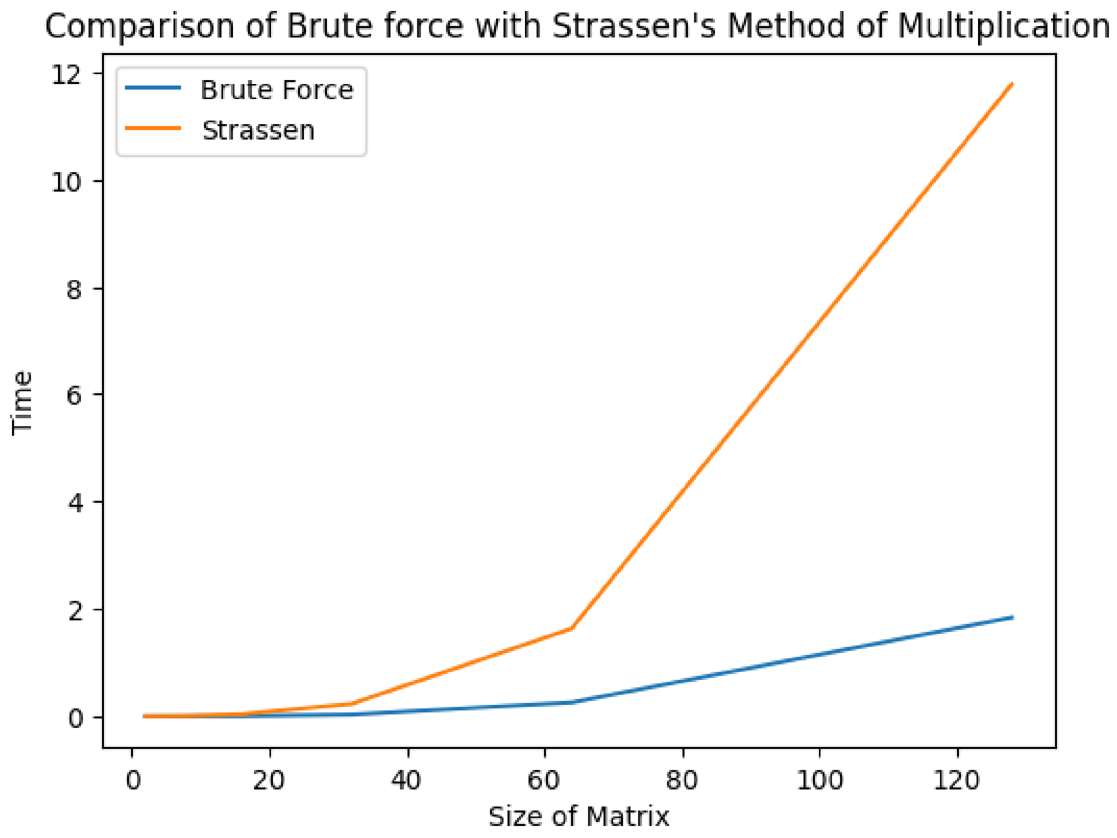
This method follows the simple way of multiplying two matrices; the column of the first matrix is multiplied to the row of the second matrix. In code we store this product of two elements in an index of a zero matrix which we initialized at the beginning adding the product element wise with the zeros in the third matrix. This way three loops are used the outer one being on the size of resultant matrix that needs to be filled and the inner two loops on the rows and columns of matrices that are being multiplied respectively. This way the complexity of this algorithm takes up to n^3 for the three loops on size n. The addition and multiplication operations take constant time so we can ignore that in our analysis for the big-Oh which comes out to be O(n^3).

5.2 Strassen's Method For Square Matrix Multiplication

Strassen's method approaches matrix multiplication differently than the aforementioned algorithm. It uses recursive divide and conquer approach and splits the matrix into smaller sub-problems. A matrix is divided into half making 4 matrices from it recursively. Then these smaller matrices are multplied using Strassen's equation which a bit odd to learn or the reasoning isn't clear.We are referencing the easy way to learn. This method works by making sure both matrices are square matrices and there size is in power of 2, this is done by augmenting/padding the initial matrix with 0's. Seeing the time complexity for this method we make 7 recursive calls instead of 8 because of the constant time subtraction, addition and copying the values for the final matrix. The number of sub-problems (a) here are 7 where the size of each sub-problem is 2 (b), while the work outside the recursion is constant 2 (c). Following this we can say $a > b^d$ is fulfilled. Master's theorem equation comes out to be $T(n) = 7T(\frac{n}{2}) + O(n^2)$, finally the complexity from master's theorem comes out to be $O(n^{log}2(7))$ which equals to $O(n^{2.81})$.

5.3 Explorations And Analysis For Matrix Multiplication

Theoretically Strassen's method performs better than the conventional naive method exponentially where naive method had the upper bound of $O(n^3)$ Strassen's method took it to $O(n^{2.81})$. However, this was not the case in our tests. The Strassen's method was considerably slower than the naive method. This was surprising, with some searching we found how Strassen's recursive calls were causing the problem here. The latency in recursive call is considerable and all the memory allocation takes away the advantage gained from the fewer arithmetic operations from the naive method. Not only that we have discovered that the run time improves for Strassen's algorithm when the input is a sparse matrix, else the recursive calls have to wait for the base case which damps the performance causing it to be slower practically.



The graph shows clearly how the presumed Strassen isn't able to outperform naive method. However, for bigger matrices this might change as the slightly less $O(n^{2.81})$ will give Strassen's some edge and more if the matrices are sparse or binary.

5.4 Brute Force For Chain Matrix Multiplication

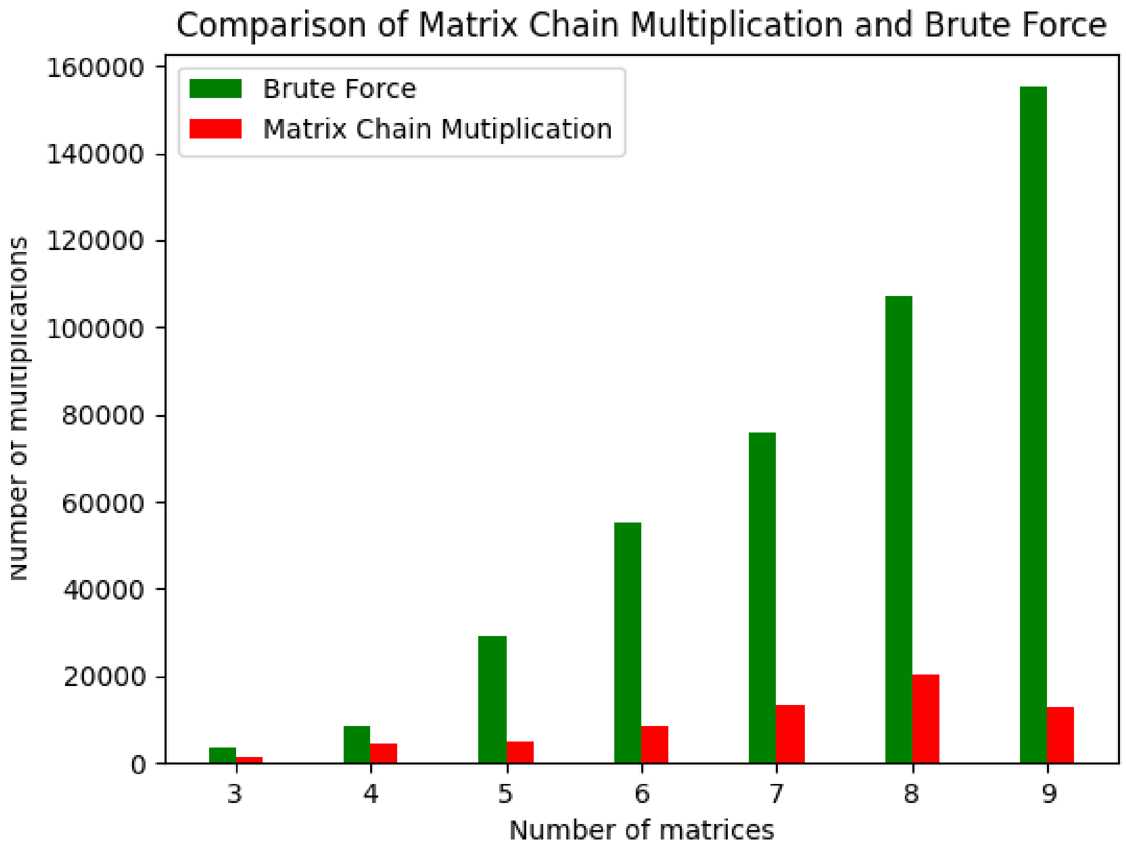
This Brute force method n matrices and multiplies them which will have random dimensions (for simplicity, the row value of a matrix is kept same as column value of the matrix before it as both should be same for multiplication to be possible). All the matrices and their dimensions are randomly generated for this multiplication. This is a computationally heavy procedure and does not guarantee optimal solution.

5.5 Chain Matrix Multiplication Using Dynamic Programming

Dynamic programming uses the memoization/tabulation to tackle the problem of repeating sub-problems. The process is similar to generating an binary search tree and tries to touch the property of optimal substructure, which ensures that all the sub-problems in the main problem are generating the optimal solutions so that when memoization/tabulation is done the resultant is already the optimal solution; this is the gist of dynamic programming. The algorithm uses $O(n^3)$ time to find the possible multiplication order which has minimum multiplication. This is same as brute force however, brute force follows the left to right order which doesn't reduce the number of multiplications. This method gives and specific order which guarantees the minimum number of multiplications.

5.6 Explorations And Analysis For Chain Matrix Multiplication

The dynamic programming approach obviously not only does faster computation but also finds the most optimum solution for the parenthesising problem using the tabulation method here. We have used the bottom up tabulation approach. We stacked both these methods against each other to find out the difference in performance and the cost at which we are able to find the fine tuned dynamic programming algorithm that finds the sequence for these notorious matrix multiplications. Following the dynamic programming approach we have already found out that it is much more faster and guarantees the best optimisation. Below given graph compares the number of multiplications in brute force with Chain Matrix Multiplication.



The graph shows clearly how the number of multiplications can be reduced if we use chain matrix multiplication

6 Conclusion And Reflections

We have seen optimal methods of matrix manipulation and operations at both matrix multiplication level and at the sequence of these multiplications being performed. We saw how Strassen's started off good with promising a lower upper bound to the problem, however, it proved to be slower due to its recursive lag. Although we did find that it works better on sparse square matrices and has a lot of use in image processing where such matrices are constantly being manipulated with different operations. Given all this Strassen's method is not preferred given its complex code and the underlying factors which makes it a tough choice.
The problem of chain matrix multiplication is optimized by the use of dynamic programming. We saw that the number of steps are drastically reduced by tabulating the sequences, then using the best possible parenthesising to multiply these matrices the least number of commutative operations for these matrices are yielded. We used the tabulation approach while there is a memoization approach which uses top down approach and has a similar upper bound.

7 References

1) Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein
2) Algorithms by Dasgupta, C.H. Papadimitriou, U.V.Vazirani
3) Geeks for geeks code for strassen method