

FA22 Algorithms for Bioinformatics, EN.605.620

Programming Project 1, Gerald McCollam – gmccoll12@jh.edu

September 27, 2022

Strassen's Method versus Brute Force Multiplication - An Analysis

1. Introduction

As part of Programming Project 1, an analysis of results obtained through comparison of two related algorithms is presented. The first algorithm is a naive or 'brute force' implementation for calculating the product of two square matrices. The second represents an enhancement to the first and is known as Strassen's algorithm.

2. Description and Justification

Strassen's algorithm (named for Volker Strassen) is a well-known improvement to a naive solution for matrix multiplication. While it is faster than the standard matrix multiplication algorithm for larger matrices, the naive equivalent is often better suited for smaller matrices.¹

An important caveat regarding application of Strassen is that the input matrices must be square and their order must be a power of two. These restrictions allow for the matrices to be split recursively into halves until a base case is reached.

(a) Data Structures

I have used standard Python data structures such as lists and dictionaries throughout my implementation. One advantage of Python in this context has been the use of list comprehensions for populating and iterating over a 2-dimensional data structure such as a matrix.

(b) Implementation

I've applied sound developmental practices in designing my code. In part, this means separating out functionality into independent modules. I also make use of Python's notebook functionality to permit recreation of my plots using Google Colab. My implementation of Strassen's can be run as a separate script, in order to test it directly.

¹See: "Strassen Algorithm". In Wikipedia, June 13, 2022.

i. Language

I was granted permission to implement this solution in Python.

ii. Modular Design

The module ‘algorithms’ implements Strassen’s and the standard algorithm; ‘analysis’ contains a Python notebook for plots; ‘data’ holds the input data and facility to create new matrices; ‘filehandler’ handles file input and output.

3. Efficiency

We refer to matrix **order** when speaking of runtime efficiency related to the size of n . A 2-row by 2-column matrix has an order of 2 ($n = 4$) while a 256x256 has an order of 256. For the naive algorithm, the triple-nested **for** loop is inefficient as the matrix order grows larger. Consider the following, where we have input matrices A ($n \times m$), B ($m \times p$) of equal order, and a third empty matrix C ($n \times p$) of appropriate size. The naive algorithm is:

```
1.   For i from 1 to n:
2.       For j from 1 to p:
3.           Let sum = 0
4.           For k from 1 to m:
5.               Set sum ← sum + A[i,k] × B[k,j]
6.           Set C[i,j] ← sum
7.   Return C
```

(a) Time Complexity

The naive algorithm’s time complexity is related to its nested loops (Lines 1,2,4). Each runs n times and does $O(1)$ work at each iteration, thus the complexity is $n * n * n * O(1) = O(n^3) * O(1) = O(n^3)$. Strassen does better, achieving sub-cubic time. The algorithm makes 7 recursive calls along with several additions and subtractions that together take cn^2 time. If we let $f(n)$ be the number of operations for a $2^n \times 2^n$ matrix then $f(n) = 7f(n-1) + l4^2$ where l depends on the number of additions and subtractions. We obtain the recurrence $T(n) \leq 7T(n/2) + cn^2$ where $T(1) = 1$. Strassen runs in time $O(n^{\log_2 7}) = O(n^{2.803})$.

(b) Space Complexity

For Strassen, we start with $3n^2$ space required for the (2) input and (1) output matrices then must allocate $3(n/2)^2$ space for recursive calls. Both the naive and Strassen algorithms are on the order of n^2 space complexity. However, Strassen may require significantly more memory compared to the naive algorithm.

4. Asymptotic Observation

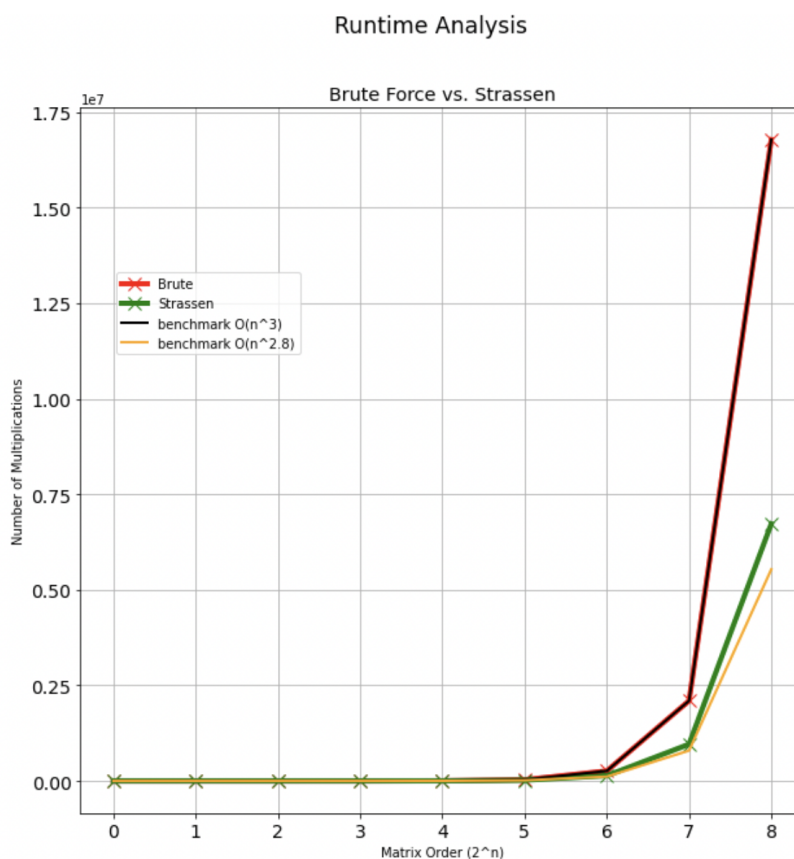
Table 1 and the plot following (next page) show the relative performance of implementations of the naive versus Strassen’s algorithm. I have gathered data where each square matrix’s order is a power of two. Note that where $n = 0$ each matrix has a single element (the base case).

Where the matrix order grows to 256 ($n = 8$) the difference in the number of multiplications required for the naive algorithm is 2.5 times that of Strassen's.

The plot below Table 1 contains four colored lines. The red is the naive or 'brute force' algorithm; the green is my Strassen's; the black is a benchmark function for n^3 ; and the yellow is a benchmark function for $n^{2.8}$.

Matrix Order 2^n	Naive algorithm	Strassen's algorithm
1	1	0
2	8	7
4	64	56
8	512	399
16	4,096	2800
32	32,768	19,607
64	262,144	137,256
128	2,097,152	960,799
256	16,777,216	6,725,600

Table 1: Number of multiplications required in Naive versus Strassen algorithms.



Note that the black and red lines are exactly the same (one covers the other) meaning that the naive algorithm precisely mimics the benchmark function. For Strassen's algorithm the results diverge, with the benchmark $f(n^{2.8})$ (in yellow) rising slower compared with my implementation of the algorithm (in green).

5. Observation versus Theory

It's not surprising that the naive implementation performs exactly as one expects it would relative to the benchmark. Whatever other runtime costs are associated with my implementation, the cubic component entirely obscures them. Both cubic and slightly-less-than-cubic functions seem to do about the same until the matrix order reaches 64 ($n = 6$). At $n = 7$ the cost of my implementation of Strassen begins to rise relative to the benchmark. Two factors may account for this: 1) the benchmark function is rounded to a power of 2.8 where in reality it is slightly greater than this; and 2) I am only counting multiplications, not the additional costs of additions and subtractions.

6. Lessons Learned

This was a challenging exercise. I struggled with my implementation of Strassen's algorithm and ultimately relied on research done on Stack Overflow. This is noted in the code. I learned much about how to use Python's command-line parser `argparse`. This is a valuable skill for developing command line applications in bioinformatics.

7. Applicability to Bioinformatics

In general, bioinformatics applications are replete with examples where matrices are a key data structure: from differential expression (RNA-Seq) experiments to integrative 'omics' studies, to basic theoretical work on genetic control. As an example, expression quantitative trait loci (eQTL) analysis links variations in gene expression levels to genotypes. eQTL analysis is a computationally intensive task that generates huge datasets; testing for association of billions of transcript-SNP (single-nucleotide polymorphism) pairs is standard practice. Such pairs may be represented as large matrices and thus manipulating these structures efficiently is a central task.[1]

Independent Component Analysis (ICA) is another example, where matrix factorization methods are used for data dimension reduction.[2] The application of ICA (Independent Component Analysis) to multiple gene expression datasets has uncovered important insights about cancer biology. In such studies, large ICA-based meta-analyses of transcriptomic data are defined by sets of metagenes that are associated with universal factors of cancer types. ICA application to transcriptomics data relies heavily on fast and efficient matrix manipulations.

Matrices are also widely applied in DNA microarray analysis, wherever a large number of gene expression profile data are produced. Efficient processing of gene expression data - often using large array data structures - is an important topic in computational molecular biology.[3]

Appendix

Please note: the following pseudocode is incomplete (Latex troubles) but is accurate in terms of my implementation. I've placed it in an appendix because I felt it didn't fit well in the main body of the report.

Algorithm 1 STRASSEN'S ALGORITHM

```
1: function STRASSEN( $A, B$ )
2:   if  $M = 1 \times 1$  then
3:     Return  $M_{11}N_{11}$  ▷ The base case
4:
5:    $M \leftarrow (A, B, C, D)$  ▷ Represent first initial matrix
6:    $N \leftarrow (E, F, G, H)$  ▷ Represent second initial matrix
7:
8:   Set  $S_1 = \text{Strassen}(B - D, G + H)$ 
9:   Set  $S_2 = \text{Strassen}(A + D, E + H)$ 
10:  Set  $S_3 = \text{Strassen}(A - C, E + H)$ 
11:  Set  $S_4 = \text{Strassen}(A + B, H)$ 
12:  Set  $S_5 = \text{Strassen}(A, F - H)$ 
13:  Set  $S_6 = \text{Strassen}(B - D, G + H)$ 
14:  Set  $S_7 = \text{Strassen}(B - D, G + H)$ 
15:
16:  Return  $\begin{pmatrix} S_1 + S_2 - S_4 + S_6 & S_4 - S_5 \\ S_6 + S_7 & S_1 + S_2 - S_4 + S_6 \end{pmatrix}$ 
17:
```

References

- [1] Andrey A. Shabalin. Matrix eqtl: Ultra fast eqtl analysis via large matrix operations. *Bioinformatics*, 28(10), 2012.
- [2] Jane Merlevede Askhat Molkenov Ainur Seisenova Altynbek Zhubanchaliyev Petr V Nazarov Emmanuel Barillot Ulykbek Kairov Captier, Nicolas and Andrei Zinovyev. Biodica: A computational environment for independent component analysis of omics data. *Bioinformatics*, 38(10), 2022.
- [3] Edward Ott Michelle Girvan Pomerance, Andrew and Wolfgang Losert. The effect of network topology on the stability of discrete state models of genetic control. *Proceedings of the National Academy of Sciences*, 106(20), 2009.