CS 124 Programming Assignment 2

Qiang Fei, Jordan Turley

April 1, 2020

1 Introduction

In this programming assignment, we explore Strassen's algorithm, which allows us to compute a matrix multiplication of two $n \times n$ matrices in approximately $O(n^{2.8})$ time, compared to $O(n^3)$ when using the regular matrix multiplication algorithm. We explore a modified Strassen's algorithm where we switch to the regular matrix multiplication algorithm once we reach a certain matrix size, and we find the optimal number for this switch. Finally, we generate random graphs and use our modified Strassen algorithm to find the number of triangles in the graph.

2 Strassen's Algorithm

The naive matrix multiplication algorithm for two $n \times n$ matrices is $O(n^3)$. However, by using Strassen's algorithm, we can reduce this to approximately $O(n^{2.8})$ as follows.

Given two $n \times n$ matrices A, B decomposed as follows:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}, B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$$

We calculate the following, which only involves 7 matrix multiplications:

$M_1 = (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2})$

$M_2 = (A_{2,1} + A_{2,2})B_{1,1}$

$M_3 = A_{1,1}(B_{1,2} - B_{2,2})$

$M_4 = A_{2,2}(B_{2,1} - B_{1,1})$

$M_5 = (A_{1,1} + A_{1,2})B_{2,2}$

$M_6 = (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2})$

$M_7 = (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})$

And we compute $C = AB$, with

$C_{1,1} = M_1 + M_4 - M_5 + M_7$

$C_{1,2} = M_3 + M_5$

$C_{2,1} = M_2 + M_4$

$C_{2,2} = M_1 - M_2 + M_3 + M_6$

and construct

$$C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}.$$

This procedure results in a time complexity of $O(n^{\log_2 7}) \approx O(n^{2.8})$, which will be better than the $O(n^3)$ algorithm as n gets arbitrarily large.

3 Modified Strassen's Algorithm

We consider a modified matrix multiplication algorithm. For n large, we begin by using Strassen's algorithm, but when we reach a certain crossover point n_0, we instead use the naive matrix multiplication algorithm, as it will be faster for smaller inputs than Strassen's algorithm. We would like to analytically find n_0, which we can do by defining functions for the number of arithmetic operations used by this modified algorithm and optimize for n_0.

For a given matrix of size $n \times n$, we can see by observing the formulas above that Strassen's algorithm takes 7 matrix multiplications and 18 matrix additions or subtractions, all of which will be on sub-matrices of size $\frac{n}{2} \times \frac{n}{2}$. This can be represented by the following recursive formula:

$$T(n) = 7T(\frac{n}{2}) + 18\left(\frac{n}{2}\right)^2$$

Next, the naive matrix multiplication takes $T(n) = n^2(2n-1)$ arithmetic operations, since for every cell in the resulting matrix, we do n multiplications and $n-1$ additions.

By combining the two, we get the following formula which determines the number of arithmetic operations:

$$T(n) = \begin{cases} 7T(\frac{n}{2}) + 18\left(\frac{n}{2}\right)^2 & n > n_0 \\ n^2(2n-1) & n \leq n_0 \end{cases}$$

That is, if the matrix size is bigger than n_0 we use Strassen's algorithm, but if it is smaller, we use the regular matrix multiplication algorithm.

Using Python, we simulated several different values of n_0 to find the one that gives us the minimum total number of operations. We simulated for matrix sizes of $n = 512, 1024, 2048$, but we should get the same value of n_0 for each. We also simulated for $n = 295, 3481057$, which were just randomly selected to make sure we get the same value of n_0 when n is not a power of two and when n is odd. The plots and results are shown below.

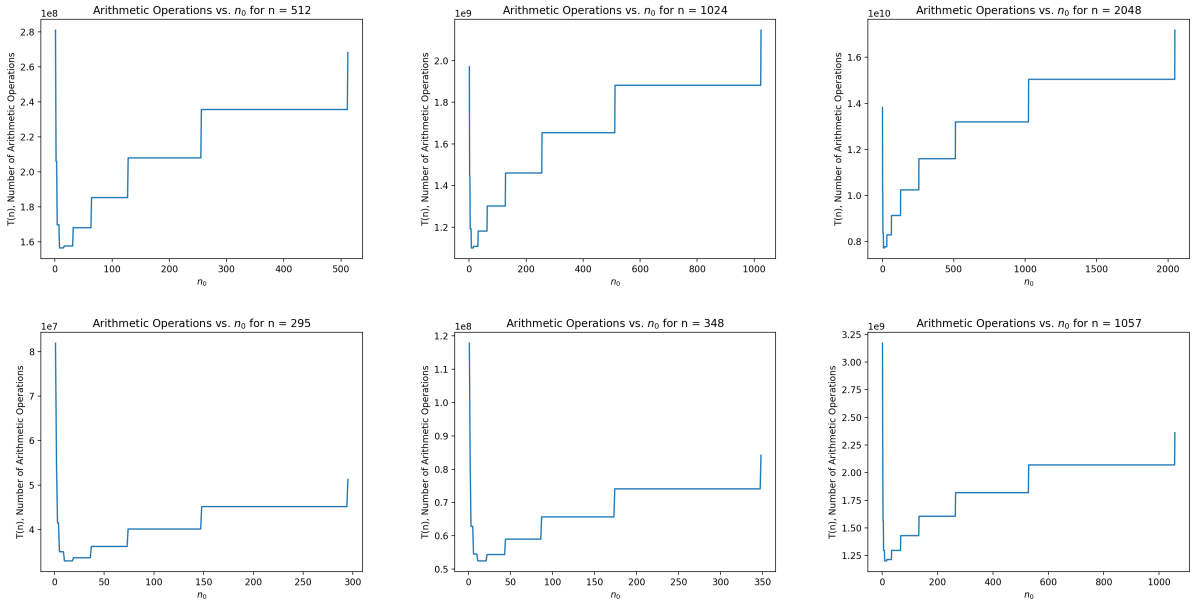


Figure 1: $T(n)$, Number of Arithmetic Operations vs. n_0 for several values of n

We see that the optimal value of n_0 is $n_0 = 8$, and is the same no matter the value of n (if n was odd $n_0 = 9$, but practically a difference this small will not make an impact). In the next part, we will determine the optimal value of n_0 in practice.

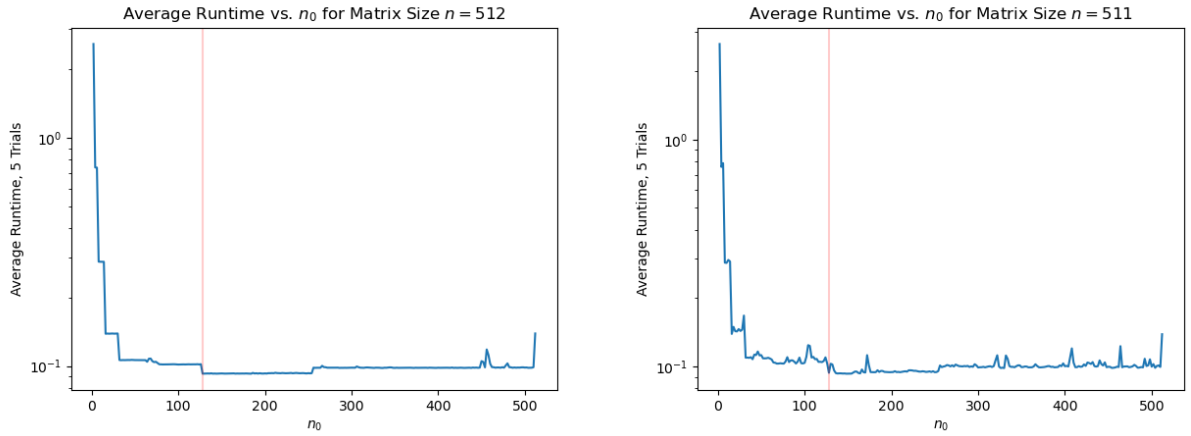
4 Modified Strassen's Algorithm in Practice

We coded Strassen's algorithm and the regular matrix multiplication algorithm in C. For Strassen's algorithm, to handle matrices where n was not even, we padded the rightmost and bottom edges with a column and row of zeros, since

$$\begin{bmatrix} A & 0 \\ 0 & 0 \end{bmatrix} \times \begin{bmatrix} B & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} AB & 0 \\ 0 & 0 \end{bmatrix}.$$

Adding a column and row of zeros will result in matrices that can be evenly blocked for Strassen's algorithm. Once we reached a matrix that was smaller than some crossover point n_0, we switched to the regular matrix multiplication algorithm.

To find the optimal value of n_0, we multiplied two matrices filled randomly with zeros and ones with $n = 512$ and $n = 511$, so that we could simulate a matrix that could easily be divided in half never using any padding, and another matrix that would require padding. We simulated values of n_0 from 2 to 512. A value of $n_0 = 2$ means we use Strassen's algorithm until we have a 2x2 matrix, and a value of $n_0 = 512$ means we do not use Strassen's algorithm at all and only use the regular matrix multiplication algorithm. We simulated each n_0 value for five trials and averaged the runtime. The results are plotted below.



(a) $n = 511$ (b) $n = 512$

Figure 2: Runtime vs. n_0 for Various Matrix Multiplication Sizes

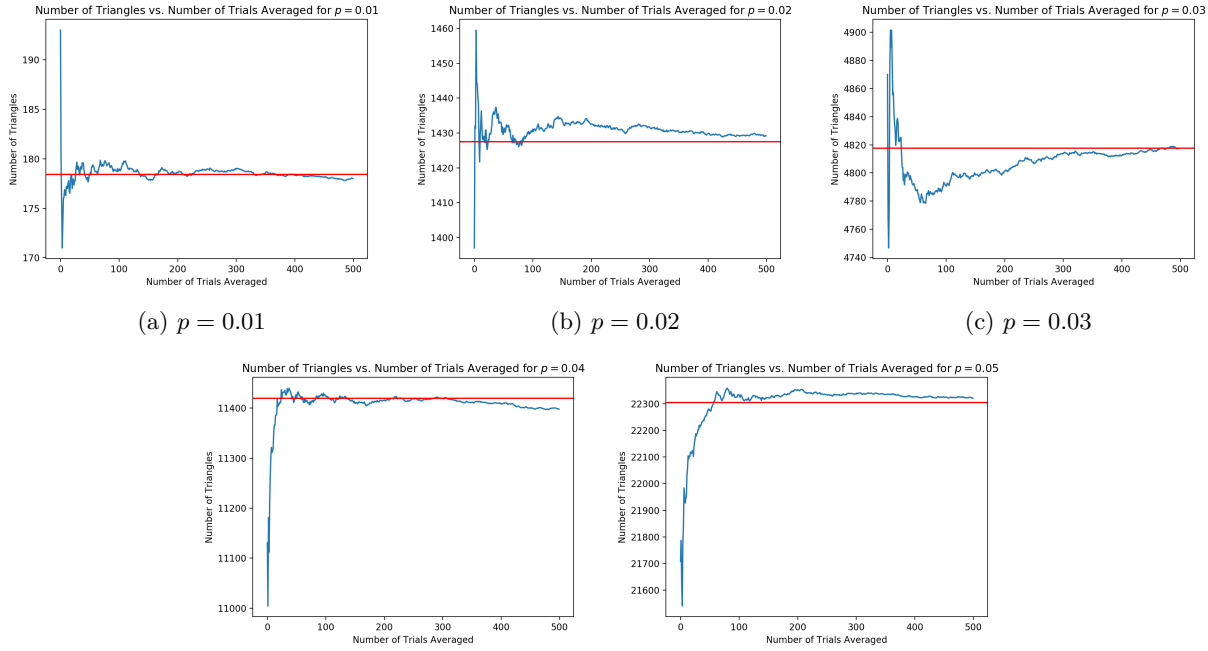
We see that the optimal value in both cases is $n_0 = 128$, as is represented on the plot by the vertical red line. For the $n = 511$ matrix there was much more variability, which we could not explain other than requiring zero-padding causing overhead that the $n = 512$ case did not require.

At first, this seems very different from our above analytical estimate of $n_0 = 8$, but when we consider the scope of matrices that this algorithm would be used for, as well as the overhead created by the actual implementation of this algorithm, this is not that big of a difference. Multiplying matrices of these sizes, i.e. dimensionality of a few hundred or a few thousand, is pretty trivial. In the real world, one may be working with matrices that have dimensionality of hundreds of thousands or millions. This is where an algorithm like Strassen's can save hours or days of computation, and by testing the algorithm on smaller cases like this, we can find the optimal n_0 that works in practice that can be extended up to much larger matrices. In addition, the implementation of this algorithm is very memory-intensive, as we have to sometimes pad the edges of a matrix with zeros which requires us to create a new matrix, in addition to creating several new matrices with every iteration of Strassen's algorithm for the calculation. This creates a lot of overhead that is unaccounted for in our earlier calculation. Finally, we did a bit of research on our own, and on the Wikipedia page for Strassen's algorithm, it says that the optimal crossover point is between 32 and 128, depending on the implementation. Our implementation is likely not as optimized as others, but a crossover point of 128 is still very reasonable.

5 Triangles in Random Graphs

As an application of the Strassen algorithm we implemented, we found the number of triangles in randomly generated graphs. To do this, we generated an adjacency matrix M for a graph G with 1024 vertices with a certain probability p of including an edge between two vertices. We calculated M^3 using our algorithm and summed the values on the main diagonal, diving by six to avoid overcounting, to determine the number of triangles in the graph. We repeated this several times and calculated an average to compare to the true average, which is $\binom{1024}{3}p^3$.

We simulated this for $p = 0.01, 0.02, 0.03, 0.04$, and 0.05, each for 500 trials. We calculated the sample average over time so that we were able to observe the convergence of the sample average to the true average, rather than just comparing the final numbers. The results are plotted and shown below, with the true average shown as a red horizontal line.



(a) $p = 0.01$ (b) $p = 0.02$ (c) $p = 0.03$

(d) $p = 0.04$ (e) $p = 0.05$

Figure 3: Sample vs. True Average Number of Triangles in Random Graph for Various Values of p

p	**Simulated Average**	**True Average**	**|Difference|**
0.01	178.022	178.433	0.411
0.02	1429.16	1427.464	1.695
0.03	4817.512	4817.691	0.179
0.04	11397.886	11419.713	21.827
0.05	22320.054	22304.128	15.925

Figure 4: Final Calculations of Sample Average vs. True Average

We see really good convergance to the true average from our simulation for each of the p values. Our final average calculation is very close to the true average, and in most cases it is very close after only a short number of trials. Our final estimates are within one or two units in each case. This simulation took about 30 minutes to run since we had to generate 500 adjacnecy matrices for each of the five p values, then calculate two matrix products for each to find M^3.

6 Conclusion

Implementing algorithms like this is very useful for a programmer, both in terms of learning this specific algorithm, but also in terms of trying to think of ways to optimize a certain procedure, even if the optimization is much more complex. Learning to code a matrix multiplication algorithm is useful, but in practice, you would likely opt to use a third party library that has already been optimized by hundreds of people for years instead of trying to code your own. The value in coding up this algorithm is to consider more complex solutions that can give you significant speedup over the more basic solutions. The basic matrix multiplication algorithm is easy to code up and only takes about ten lines and five minutes of your time, but it will be slower when compared to an algorithm like this, which is much more complex. Due to all of the sub-matrices and memory management, this algorithm took us about 150 lines of code and a few hours to code up. This still isn't a lot but is significantly more than the basic matrix multiplication. However, this algorithm will be significantly faster for large matrices. Since we only ran up to $n = 2048$ we only saw differences of a few

seconds, but it would be interesting to compare the two algorithms for a really big matrix, like $n = 10000$ or $n = 100000$. The difference here would likely be surprising.

It is also useful to think about combining different algorithms to give the best possible runtime. In theory, we would always prefer Strassen's algorithm since it's complexity is smaller, but in practice, as we saw above, Strassen's algorithm creates so much overhead that for smaller matrices, it is much faster to use the regular algorithm. Because of this, we got the optimal runtime by switching to the basic algorithm once we reached some crossover point. Skills like this are really useful for someone planning on working as a programmer, because your boss likely doesn't care about the theoretical complexity, but cares about making the code run as fast as possible on whatever input you give it. Knowing to switch to the simpler algorithm could save days or weeks of computation time down the road.

In terms of our algorithm, there are definitely optimizations that could be made, but we were able to see the usefulness of this algorithm even with our somewhat unoptimized version. A lot of the computational time was spent creating or freeing up memory. If we could optimize this, it may have changed our crossover point to a smaller value which could have made our algorithm even faster.