CS124 Coding Project 2 Write-up

Luke Bailey, Matthew Karle

February 2020

1 Introduction

Improving the running time of matrix multiplication is a popular and widely studied topic in theoretical computer science. The first and most familiar algorithm to do so is Strassen's algorithm, which defines a lengthy set of operations that ultimately reduce the recursion of a divide and conquer multiplication implementation from eight calls down to seven. Although this has a significant impact on asymptotic bound for run time, moving from a naive $O(n^3)$ to Strassen's $O(n^{\log_2(7)}$, the algorithm is often dismissed as impractical for any reasonable matrix dimensions due to the cost of the added operations that circumnavigate the eigth recursive call.

Yet the fact remains that Strassen's is asymptotically the faster algorithm than naive matrix multiplication. Then for very high matrix dimensions, one would be better off using Strassen's, though as it reduces the inputs of its recursive calls to smaller matrix dimensions, one would be better suited switching to naive multiplication to complete the call. That is, there exists some dimension n_0, at which naive matrix multiplication is faster than another Strassen call. In this paper, we seek to analytically and empirically determine this n_0, such that it optimizes the speed of a Strassen's/Naive hybridization for arbitrary matrix size inputs. We then sought to use this hybridization and the empirically determined n_0 to address other problems involving matrix multiplication.

2 Analytical n_0

We can find an estimate for the value of n_0 by assuming all arithmetic operations have a cost of 1. We know the cutoff should be in place when using the standard matrix multiplication algorithm is less costly than applying one level of the Strassen recursion followed by standard matrix multiplication. Let $M(n)$ be the cost of the standard matrix multiplication algorithm on a square matrix of size $n \times n$. The standard matrix multiplication algorithm performs n^3 multiplication operations and $n^3 - n^2$ addition operations on an $n \times n$ matrix. Thus we can conclude that $M(n) = 2n^3 - n^2$. The following analysis concerns matrices of even sizes. We will address the odd case after.

2.1 Even n

Now we must calculate the cost of applying a single level of Strassen's recursion followed by standard matrix multiplication. The Strassen recursion would split two $n \times n$ matrices into 8 $\frac{n}{2} \times \frac{n}{2}$ matrices and perform the following operations:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

$$
\begin{aligned}
p_1 &= A(F-H) & AE+BG &= p_3+p_4-p_2+p_6\\
p_2 &= (A+B)H & AF+BH &= p_1+p_2\\
p_3 &= (C+D)E & CE+DG &= p_3+p_4\\
p_4 &= D(G-E) & CF+DH &= p_5+p_1-p_3-p_7\\
p_5 &= (A+D)(E+H) & & \\
p_6 &= (B-D)(G+H) & & \\
p_7 &= (A-C)(E+F) & &
\end{aligned}
$$

By performing one level of Strassen's recursion followed by standard matrix multiplication, we will have to perform 7 standard matrix multiplications each of cost $M(\frac{n}{2})$ (to calculate p_1 to p_7) and 18 additions/subtractions (in the calculation of p_1 to p_7 and the matrix outputs $AE+BG$ to $CF+DH$). If addition and subtraction of individual numbers is unit cost, then addition and subtraction of two $\frac{n}{2} \times \frac{n}{2}$ matrices costs $\frac{n^2}{4}$. Let $S_1(n)$ equal the cost of applying one level of Strassen's recursion followed by standard matrix multiplication on two $n \times n$ matrices. From the above we can form:

$$S_1(n) = 7M(\frac{n}{2}) + \frac{18n^2}{4}$$

Substituting $M(n) = 2n^3 - n^2$ in we form:

$$
\begin{aligned}
S_1(n) &= 7(\frac{n^3-n^2}{4}) + \frac{18n^2}{4}\\
&= \frac{7n^3+11n^2}{4}
\end{aligned}
$$

We would like the calculate the values of n for which $M(n) \leq S_1(n)$. The maximum value of the solution to this inequality will be our analytical cutoff point.

$$
\begin{aligned}
M(n) &\leq S_1(n)\\
2n^3 - n^2 &\leq \frac{7n^3+11n^2}{4}\\
n^3 - 15n^2 &\leq 0\\
n^2(n-15) &\leq 0\\
n - 15 &\leq 0\\
n &\leq 15
\end{aligned}
$$

The maximum value of n from the above inequality is the point at which standard matrix multiplication has the same cost as applying a single level of Strassen's recursion followed by standard matrix multiplication. Our analytical cutoff point, n_0 is this value, thus $n_0 = 15$ by this analysis, when n is even.

2.2 Odd n

In many (and our) implementation of Strassen's algorithm with a cutoff, at any point of the recursion, an odd sized matrix is padded with 1 column and 1 row of zeroes to make the size of the matrix even. In our above anaysis, because we are only considering a single application of the Strassen's recursion, an odd sized matrix would be padded once. If we consider the cost of this padding to be 0 (or insignificantly small in comparison to other operations), we can padd an odd sized matrix and then apply the even analysis from section 2.1. This would once again give us an n_0 value of 15.

If however we assign a unit cost for each padded 0, the operation to pad an $n \times n$ matrix would cost $2n + 1$. Thus we modify the analysis from 2.1 to form the below inequality:

$$2n^3 - n^2 \leq \frac{7n^3 + 11n^2}{4} + 2n + 1$$
$$n^3 - 15n^2 \leq 8n + 4$$
$$n^3 - 15n^2 - 8n - 4 \leq 0$$
$$n \lessapprox 15.53$$

This means, when $n = 16$, $M(n) > S_1(n)$ but when $n = 15$ $M(n) < S_1(n)$ and thus we once again have a cutoff point of $n_0 = 15$.

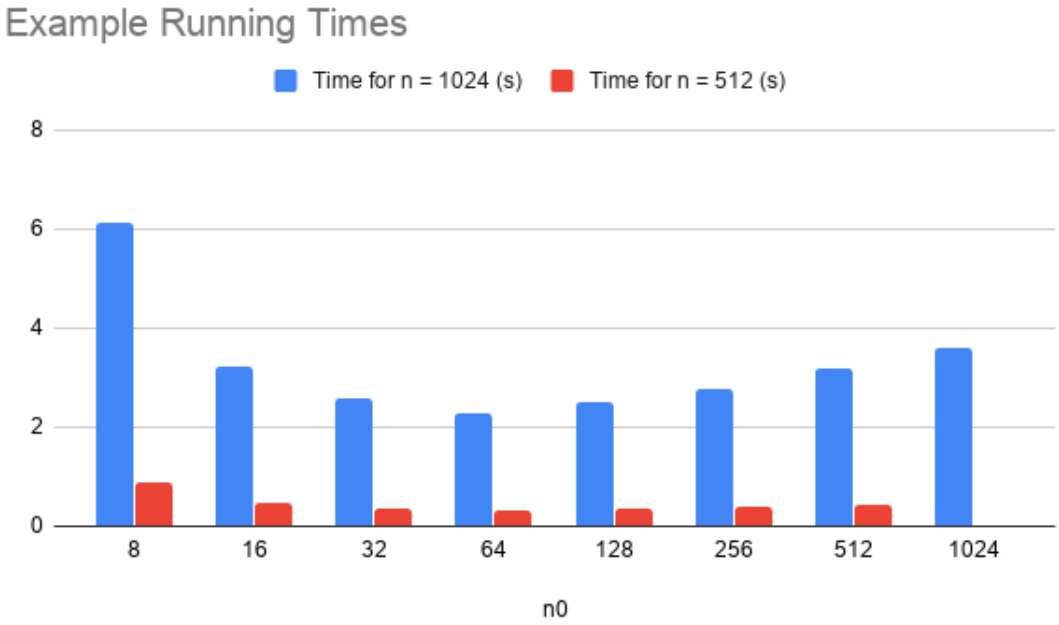
3 Experimental n_0

3.1 Results

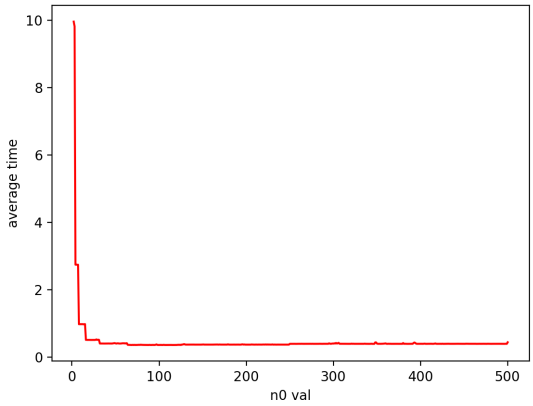
The analytical determination of n_0 was predicated on assumptions regarding the cost of implementations of various key operations in both Strassen's and naive matrix multiplication algorithms. Therefore, in order to determine a realistic n_0 for our implementations of those algorithms, we would have to run a variety of tests and find an empirical result.

Beginning with only powers of two as our matrix dimension inputs, we tested the possible n_0 values for minimum algorithm run time and determined a clear optimum at 64. This seemed a plausible deviation from our analytical result accounting for the actual costs in practice of the underlying operations of our algorithms, and it made sense intuitively that times would be suboptimal on either side of $n_0 = 64$, as crossovers smaller than this would begin naive matrix multiplication very late at $n = 32$, and crossovers larger than this would begin naive matrix multiplication too soon at $n = 128$, when an additional strassen's would evidently be optimal.

n_0	Time for $n = 1024$ (s)	Time for $n = 512$ (s)
8	6.125486	0.878308
16	3.217259	0.456887
32	2.575789	0.368715
64	2.297446	0.334688
128	2.498867	0.369846
256	2.755151	0.40237
512	3.176167	0.450471
1024	3.595777	N/A

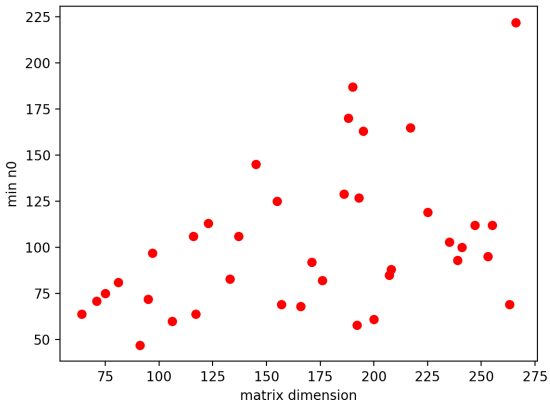


Therefore, at this point, we felt that $n_0 = 64$ was our substantiated empirical result for crossover. However, upon expanding input domain from powers of two to all positive integers, this became less clear.



The test with results pictured above run on arbitrary input size $n = 500$ for all $2 \leq n_0 \leq 500$ showed a steep improvement in run time up until $n_0 = 64$, after which there was a relative plateau up to and through $n_0 = 128$. This made sense, and suggested that all crossovers in this range were not bad approximations for the optimal. However, exploring this plateau region further, we found that the best crossover was not necessarily - and in fact was rarely - exactly 64 for an arbitrary input.

To better examine the optimal then, we collected the n_0 value that performed best for each dimension size from $32 \leq n \leq 256$. Each dimension size would perform 5 trials over randomized matrices for each n_0 in an attempt to reduce random noise. However, our results were still noisier than we originally anticipated.

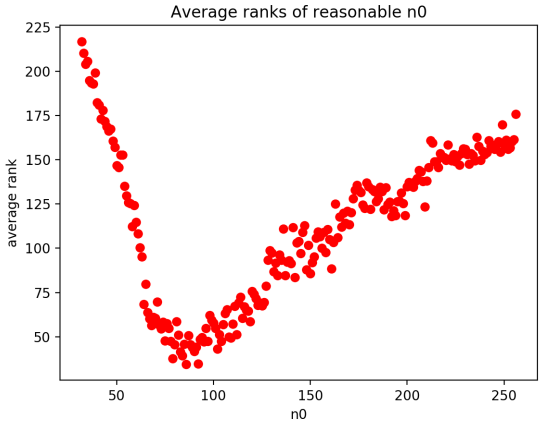


Two things were clear upon early examination of the data. When the input size was less than 64, the minimum $n0$ was consistently equal to the sample size - that is, there was perfect one to one correlation between input size and cutoff. After 64, the graph loses such clear correlation, showing a muddled set of results that bounce around between 64 and 128 without clear correlation to input. However, very few exceeded 128 - as the graph shows, only six or so anomalies proposed a cutoff to naive multiplication higher than 128.

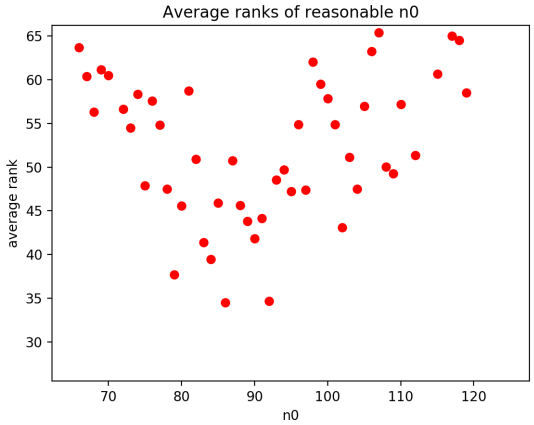
With some thinking, we determined the intuition for this variation within $64 \leq n_0 \leq 128$. This followed from the inclusion of arbitrary even and odd inputs. All such inputs greater than 128 would necessarily half to a recursively called input in the range $64 \leq n \leq 128$; however, they would not necessarily half to 64 itself. Then for some inputs, to have a cutoff at 64 would more practically mean a cutoff well below 64 and closer to 32, which our graph at input $n = 500$ had already shown to be an exponentially worse set of cutoffs. An input that eventually halved to $n = 66$, for example, would perform another, suboptimal round of strassen's to get down to $n = 33$ if it had a cutoff at $n_0 = 64$, whereas a cutoff at $n_0 = 128$ would begin its naive matrix multiplication at $n = 66$. Then in spite of nominally having a better cutoff all the way at $n_0 = 128$, the algorithm would actually begin its naive multiplication essentially at $n = 64$. This explained why $n_0 = 64$ was rendered impractical when considering arbitrary integer inputs.

Following this logic might lead to the conclusion that the optimal cutoff for a given n is its recursive call input that lies between 64 and 128, since these bounds behave so sporadically for arbitrary inputs. However, we were looking for a single constant to optimize crossover for all arbitrary input sizes. This would need to be one that could correctly balance the goal to begin naive multiplication closer to 64 without overshooting as a hard cutoff at 64 exactly would do.

To find such a value, and moreover confirm the relative similarity of performance within our range, we decided to extend our minimum n_0 data collection from before to instead record the performance of each $32 \leq n_0 \leq 256$ value, assigning that n_0 a performance rank from first place to 224th associated with that input size, and then averaging each n_0's ranks over all tested arbitrary inputs. This way, if a cutoff was always first for all inputs, for instance, it would generate an average rank of one. Below is the graph of average ranks for all n_0.



This, we felt, was our most revealing set of results. Outside of $64 \leq n_0 \leq 128$ were ubiquitously poor results north of 100 for average rank - these n_0 were consistently in the bottom half for performance. The steep crevasse within that range affirms the high performance of those n_0. Zooming in only within those bounds shows reasonably similar performance, consistently within the top 30th percentile and with less dramatic rates of change for contiguous n_0 values.



However, there is still a fairly clear shape to the graph, reaching its best ranks in the 80s and 90s, with the top performer being $n_0 = 86$ with an average rank of 34.5. Based on our established intuition from before and these results, we feel that this represents an appropriate n_0 for our implementation. That is, this cutoff is well positioned between 64 and 128 so as to not begin naive multiplication too early, as might be the case with $n_0 = 128$, but also not drastically overshoot and begin naive disastrously late in the 30s as can happen with n_0 closer to 64. While this will not necessarily be best for every input, it represents the best n_0 to handle any arbitrary input size regardless of where its recursively called inputs lie in the clearly bounded cutoff range of $64 \leq n \leq 128$.

3.2 Comparison with Analytical n_0

As was briefly mentioned above, the analytical n_0 value we found is lower than the empirical data suggests it should be. This can however be attributed to the fairly simple model we used to calculate the analytical n_0. All additions, subtraction and multiplications do not in fact take the same time to run on a real machine and other considerations have to be taken into account when implementing an algorithm in real life. For example costs of storing and retrieving information (which can vary greatly according to the cache friendliness of how an algorithm

is written which is explored later in this paper). Thus it intuitively makes sense that with the added complexities of implementing Strassen's algorithm on a real machine we would find a larger n_0 value than we theoretically calculated.

4 Triangles In Graphs

As described in the programming assignment we used 1024×1024 matrices to generate random unweighted graphs by assigning if an edge existed according to various different probabilities for each run of the test. We then found the number of triangles in these graphs using the method described in the assignment.

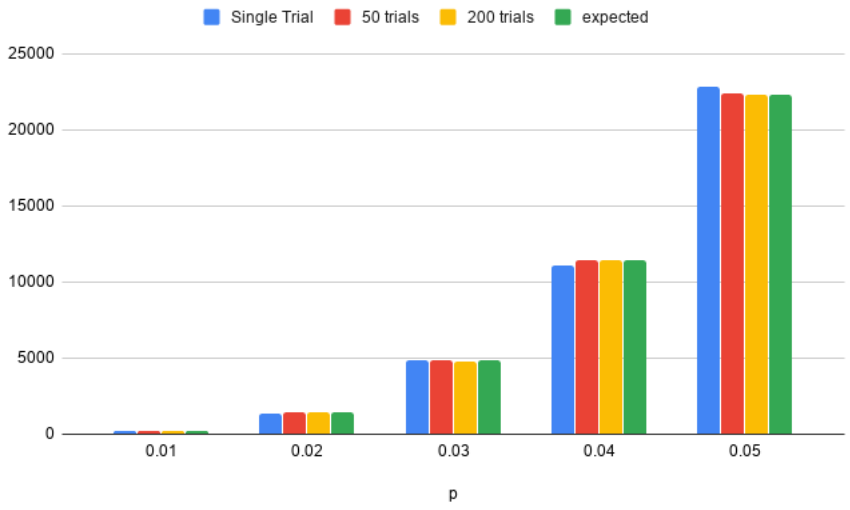
4.1 Results

Our results are below. On each run of the test, we changed the probability that an edge existed in the graph between two vertices. We have results for single runs of the test, and averages over multiple runs of the test. The expected value for numbers of triangles in these graphs is $\binom{1024}{3}p^3$ where p is the probability that for a given graph, an edge exists between any two vertices.

p	Single Trial	50 trials	200 trials	expected value
0.01	167.00	178.70	179.57	178.43
0.02	1384.00	1418.64	1428.59	1427.46
0.03	4893.00	4825.98	4799.21	4817.69
0.04	11111.00	11402.24	11421.50	11419.71
0.05	22885.00	22378.62	22307.76	22304.13

4.2 Comparison with expected number of triangles

We can see from the above data that the values generated by our algorithm were very similar to the expected values. This can be seen slightly clearer in the below chart of our data.



As expected, because we are randomly assigning if edges exist according to a specific probability, the more trials that were the run, the closer the average got to the expected value. With 200 trials and $p = 0.05$, our average number of triangles was only 3.63 off the expected, an error of only 0.015% which can simply be attributed to the

random nature of the experiment. The chart also shows the cubic nature of the expected number of triangles which our experimental outputs follow.

5 Project Discussion

5.1 Implementation and optimisations

We began by implementing Strassen's using python. As expected this was quick to implement but my partner and I were unhappy with how fast it ran. We tried to optimise it (ensuring there was no copying of large matrices etc.) and this worked to an extent, but we knew we could get far better results if we used a lower level programming language. This led us to switch to a C implementation. Once we had completed this, it was indeed far faster than our python implementation. We spent some time optimising it in a number of ways.

Firstly, it was incredible to discover the efficiency gains to be had when we used cache friendly orders of for loops in our algorithm. After some online research we became aware of the importance of harnessing cache locality. The biggest gain came from reordering the for loops we used for standard matrix multiplication to ensure we were best using locally cached pieces of data.

We also always made meticulous use of pointers throughout the coding process. We were aware of the inefficiencies of copying large data structures when calling functions with and returning said data structures. This meant we ensured we were always passing and returning pointers to our large matrices as opposed to passing the matrices themselves.

We were also aware that unnecessary memory allocation and deallocation was a source of possible losses in efficiency. We were thus careful to avoid this. Take for example the memory allocation required for a single level of recursion of the Strassen's algorithm.

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} C1 & C2 \\ C3 & C4 \end{bmatrix}$$

$$\begin{aligned} p_1 &= A(F-H) \\ p_2 &= (A+B)H \\ p_3 &= (C+D)E \\ p_4 &= D(G-E) \\ p_5 &= (A+D)(E+H) \\ p_6 &= (B-D)(G+H) \\ p_7 &= (A-C)(E+F) \end{aligned} \qquad \begin{aligned} C1 &= p_3 + p_4 - p_2 + p_6 \\ C2 &= p_1 + p_2 \\ C3 &= p_3 + p_4 \\ C4 &= p_5 + p_1 - p_3 - p_7 \end{aligned}$$

We have to allocate memory for sub-matrices A, B, C, D, E, F, G, H and p_1 to p_7. However we discovered it was not necessarily to allocate memory for $C1$ to $C4$ because at this stage of Strassen, it no longer needs any of the data stored in A, B, C, D, E, F, G, H and thus we can reuse A, B, C, D for $C1$ to $C4$, thus avoiding unnecessary memory allocation and deallocation.

The final area in which we greatly optimised our algorithm was the implementation of padding. After getting the algorithm to work for matrices whose sizes were powers of 2, we moved onto adapting it so it would work for inputs of any size. To do this we chose to pad matrices with 0's. We first considered padding any input whose size was not a power of two up to the next power of two. We decided not to do this however because it seemed innately inefficient. Take for example a 129 size input matrix. This would have to be padded all the way up to a 256×256 matrix which would take non-trivial time, and then our Strassen's algorithm would have to run on this larger Matrix. We chose to instead detect at any level of the recursion of the algorithm if the Matrix being operated on was of odd size and if so pad it by a single row and column to make it of even size. This greatly reduces the amount of padding that has to be done and thus poses a great efficiency gain over the alternate solution.

5.2 What we learned from the assignment

My partner and I learned a great deal from this programming assignment. We were particularly surprised and interested to learn about the great importance of writing cache friendly algorithms and how massively this can improve or slow running times. Additionally, previous to this assignment we both understood that python was slower than other languages like C, but did not fully appreciate to what extent. Now looking back and comparing our python and C implementation running times, we appreciate this difference in efficiency far more.

Although C was substantially faster, and our experience using it in programming assignment 1 had it fresh in our memory, we did run into some tricky language specific issues. Debugging segment faults in such a complicated recursive algorithm proved an engaging challenge. We took to using valgrind to illuminate the location of these issues, and simultaneously discovered a series of memory leaks going unattended. Resolving these issues taught us a great deal about the nature of low level languages like C and memory.

The last major learning experience outside of our implementation itself was with regard to data collection and presentation for our empirical n_0. Unlike the previous programming assignment, it was not immediately cut out for us what data merited recording and demonstration for the reader. To make this process easier, we wrote a python script to call our strassen program for us and collect its output. This easily permitted the creation of complicated graphs which we felt were essential to our understanding and detailing of the results. Deciding what exactly to use this python script for was also a huge learning experience. As we further explored the results, it became more and more apparent that there was not necessarily an obviously proven optimal n_0, but rather a great deal of variation that needed explanation and exposition of implementation. We built upon each iteration of data collection to eventually formulate what we felt was a thorough and conclusive set of information permissive to our argument. This experience was a good one beyond the scope of the assignment and even beyond that of computer science. It was extremely illuminating about the general principles of empirical experimentation and argument.