CS 124 Programming Assignment 2: Spring 2023

Your name(s) (up to two): Gabriel Chiong

Collaborators: None

No. of late days used on previous psets: 0
No. of late days used after including this pset: 0

This document contains the accompanying write-up to my implementation of Strassen's divide and combine matrix multiplication algorithm for n by n matrices. The goal of this experiment was to determine the optimal crossover point at which the base case of the recursive Strassen's algorithm is invoked. Specifically, the base case uses the conventional method of matrix multiplication on the subproblems of Strassen's divide and combine strategy. In the context of this experiment, the optimal crossover point is defined to be the matrix dimension size that allows Strassen's algorithm to outperform the running time of the conventional matrix multiplication method. This experiment includes both analytical and experimental analysis when comparing the running times of both methods.

Our implementation of Strassen's matrix multiplication in this experiment also provides an efficient method to count the number of triangles in random graphs. A triangle is defined to be a length-3 cycle in an undirected graph. As part of this experiment, we also investigate the relationship between the number of triangles in a random graph and the probability of including an edge in the graph. Using Strassen's matrix multiplication speeds up the computation of counting random triangles, as will be detailed in further sections of this document.

Overview

The complete implementation for this experiment is separated into the main program (`main.cpp`), and the two sets of header and implementation files (`strassen.h`, `strassen.cpp`, `random_triangles.h`, and `random_triangles.cpp`). There is also a bash script `time.sh` to automate the collection of running times and crossover points for a number of trials, a `Makefile` to automate the compilation of these different files, and a suite of unit tests to verify correctness of the implementation on some toy examples (`tests.cpp`). Some of these unit tests are carried out on matrices with randomly generated entry values (Strassen's method versus the conventional algorithm), and others test our implementation's correctness against open-source libraries for matrix multiplication, such as Python's `numpy` library. The instructions to execute the program can be found in the file `README.md`.

Separate to this is a directory called `analytical_analysis/` which contains my analytical analysis to determine the crossover point, and uses a simple Python program to numerically check this (`numerical_crossover.py`). There is also a directory which contains random input matrices for unit testing called `test_input/`. It contains both the input values for the matrices to be multiplied, and the expected result of the matrix multiplication. A wide range of matrix dimensions and entry values are included for testing.

The design choices for this experiment will be elaborated upon in further detail in later sections, but a summary of our considerations are provided here as reference. Our implementation uses pass-by-reference functions whenever possible to minimize the amount of memory allocation and deallocation. Pointers (or indices) into matrices are used to minimize the amount of memory allocation and deallocation. Also, writing cache-friendly code by exploiting locality of reference is used to optimize matrix operations. Lastly, the padding strategy to handle matrix dimensions that are non-powers of two is carefully considered, although this experiment does not focus on the details of comparison between different padding strategies.

Analytical Crossover

Mathematical Analysis

We begin by using analytical methods to determine the optimal crossover point for Strassen's matrix multiplication algorithm. We carry this part of the investigation out in two stages: one where we assume that the dimension of the matrix n is an even number, at least until it reaches the crossover point, and the other n is odd. The accounting model of our analytical analysis attributes a cost of 1 to any arithmetic operation (adding, subtracting, multiplying, or dividing two real numbers), and all other operations are free. We will now attempt to find the optimal size n_0 to use for Strassen's algorithm using empirical means, such that the number of operations will be less than that of the conventional one for sufficiently large original problem sizes.

The number of arithmetic operations to compute the conventional matrix multiplication is $n^2(n+n-1) = n^2(2n-1)$. The n^2 term comes from the number of entries of each matrix, and each entry is the dot product of a row and column of the input matrices, which are n integer multiplications and $n-1$ additions (and subtractions). On the other hand, Strassen's algorithm for matrix multiplication can be defined by the recurrence $T(n) = 7T(n/2) + 18(n/2)^2$. This is because the algorithm recursively divides the original matrix into sub-matrices of dimension $n/2$, and uses a further seven recursive calls. Apart from the recursive calls, there are a total of 18 pairwise matrix addition and subtraction operations, each of which is applied on a matrix with $(n/2)^2$ entries.

Since we seek to determine the crossover point where the number of operations of the conventional algorithm is greater than the number of operations by Strassen's algorithm, we can write this mathematically as follows:

$$\begin{aligned} n^2(2n-1) &> 7T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 \\ n^2(2n-1) &> 7\left(\frac{n}{2}\right)^2\left(2\left(\frac{n}{2}\right)-1\right) + 18\left(\frac{n}{2}\right)^2 \\ 2n-1 &> \frac{7}{4}(n-1) + \frac{18}{4} \\ n &> 15 \end{aligned}$$

We can thus conclude that the matrix dimension at which the conventional algorithm begins to exceed that of Strassen's algorithm in terms of the number of operations is when $n > 15$, so our crossover point is $n_0 = 16$. However, we have made an important implicit assumption about n in our mathematical analysis, namely that it is an even number. To determine the crossover point for n as an odd number, we can pad the input matrices with a row and column of zeroes, so the input matrices have a dimension of $(n+1) \times (n+1)$. As we did in the case when n was even, we again write this mathematically as follows:

$$\begin{aligned} n^2(2n-1) &> 7T\left(\frac{n+1}{2}\right) + 18\left(\frac{n+1}{2}\right)^2 \\ 2n^3 - n^2 &> 7\left(\frac{n+1}{2}\right)^2\left(2\left(\frac{n+1}{2}\right)-1\right) + 18\left(\frac{n+1}{2}\right)^2 \\ 8n^3 - 4n^2 &> 7n(n+1)^2 + 18(n+1)^2 \\ n^3 - 4n^2 &> 32n^2 + 43n + 18 \\ n^3 - 36n^2 - 43n - 18 > 0 \\ n &> 37.17 \text{ (approx.)} \end{aligned}$$

Where we obtained the final line using a free online numerical computing tool and verified with hand calculations after. We can therefore conclude that in the case that n is odd, the matrix dimension at which the conventional algorithm begins to exceed that of Strassen's algorithm in terms of the number of operations is when $n > 37.17$, so our crossover point is $n_0 = 38$ when n is odd.

Numerical Analysis

To confirm our results from our mathematical analysis, we wrote a simple program in Python to calculate the number of operations using the same equations, and trialed it on a range of values for n. The number of operations for the conventional matrix multiplication algorithm was calculated in the straightforward way as before, and the number of operations for Strassen's algorithm was calculated using a recursive function, based on the recurrence stated in the previous section. We compare the number of operations needed by the conventional matrix multiplication, Strassen's matrix multiplication with a crossover $n_0 = 16$ when n is even and $n_0 = 38$ when n is odd (let us call this "Crossover Strassen" for brevity), and another variant of Strassen's algorithm which only invokes the base case at $n_0 = 1$ (let us call this "Pure Strassen" for brevity). The results are presented below in Table 1, where the last column states whether Crossover Strassen uses less operations than the other two methods.

From the Table 1, it is clear that our mathematical analysis appears to agree with our numerical analysis. For Pure Strassen, it easily takes the most number of operations starting at $n = 2$ and remains higher consistently by what seems to be an increasing rate. Comparing the conventional matrix multiplication to Crossover Strassen, the number of operations remain equal from $n = 2$ to $n = 17$ since these values of n are below the crossover point for the Crossover Strassen algorithm. However, once we reach $n = 18$, the crossover point defined for even numbers in the Crossover Strassen algorithm, we see that the number of operations for conventional matrix multiplication is higher (11340 versus 11097). From the table of results, it can be observed that the conventional algorithm no longer performs better than the Crossover Strassen algorithm for even numbers. It is a similar situation for odd numbers, except the crossover point is when $n = 38$, and from then on, the conventional algorithm does not do better than the Crossover Strassen algorithm for any number of n. We provide Figure 1 below to highlight the observed trends until $n = 200$ which seems to agree with our findings.

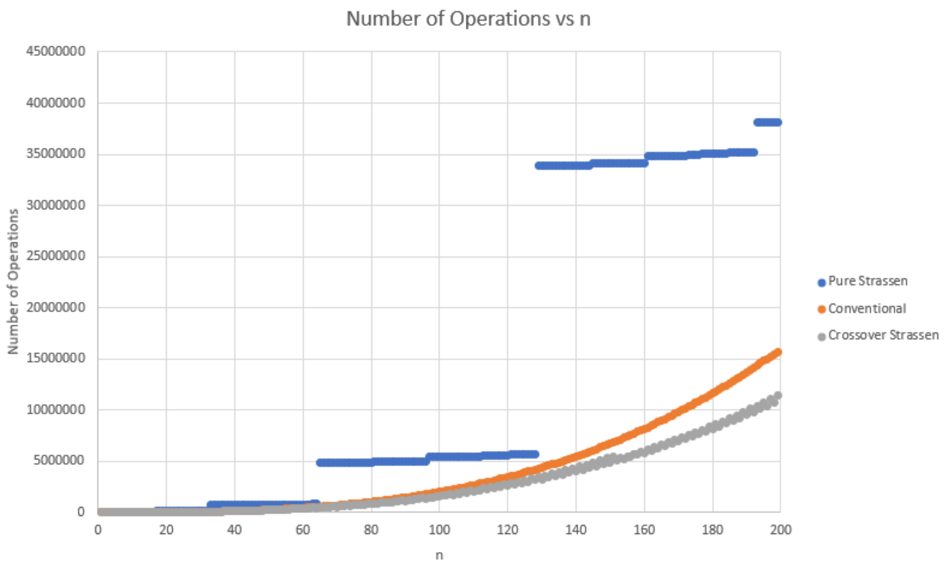

Figure 1: Comparing the number of operations until $n = 200$.

n	Pure Strassen	Conventional	Crossover Strassen	Crossover Strassen Faster?
2	25	12	12	No
3	247	45	45	No
4	247	112	112	No
5	1891	225	225	No
6	1891	396	396	No
7	2017	637	637	No
8	2017	960	960	No
9	13687	1377	1377	No
10	13687	1900	1900	No
11	13885	2541	2541	No
12	13885	3312	3312	No
13	15001	4225	4225	No
14	15001	5292	5292	No
15	15271	6525	6525	No
16	15271	7936	7936	No
17	97267	9537	9537	No
18	97267	11340	11097	Yes
19	97609	13357	13357	No
20	97609	15600	15100	Yes
21	99373	18081	18081	No
22	99373	20812	19965	Yes
23	99787	23805	23805	No
24	99787	27072	25776	Yes
25	108049	30625	30625	No
26	108049	34476	32617	Yes
27	108535	38637	38637	No
28	108535	43120	40572	Yes
29	110947	47937	47937	No
30	110947	53100	49725	Yes
31	111505	58621	58621	No
32	111505	64512	60160	Yes
33	686071	70785	70785	No
34	686071	77452	70949	Yes
35	686701	84525	84525	No
36	686701	92016	83511	Yes
37	689761	99937	99937	No
38	689761	108300	97470	Yes
39	690463	117117	107991	Yes
40	690463	126400	112900	Yes
41	703549	136161	125234	Yes
42	703549	146412	129874	Yes
43	704323	157165	144222	Yes
44	704323	168432	148467	Yes

Table 1: The number of operations needed for two variants of Strassen's and the conventional algorithm.

Experimental Crossover

In this section, we first discuss some of the key ideas and implementation considerations when writing our program for this experiment. This includes software design choices and optimizations of both the conventional matrix multiplication algorithm and Strassen's divide and combine algorithm. Following the implementation overview, we proceed to present the results of our experiment.

Implementation Considerations and Optimizations

One idea for practical improvement to our program was to use pointers (indices) into matrices to carry out operations, rather than initializing a smaller sub-matrix with identical values. Specifically, we aimed to minimize the amount of memory allocation and deallocation our program uses, by operating on the original matrix using pointers to delineate the extent of the sub-matrix operation. This is in contrast to an alternative approach which initializes a smaller matrix, copies over the relevant entries from the original matrix, carries out the desired operations on the smaller matrix, then copies over the result entries back into the original matrix. Although we could not figure out a way to completely do without allocating temporary sub-matrices in our program, we minimized it as much as possible. This idea was used for all matrix operations, including both Strassen's algorithm and the conventional algorithm.

An efficiency-related idea we implemented for our program in general was to use pass-by-reference functions where possible, in order to reduce the amount of copying large blocks of data. For matrix operations such as addition, subtraction, and the conventional multiplication method, matrices are passed by reference, including both the input matrices and the result matrix. For Strassen's algorithm, the same idea is used, so both multiplicand matrices and the result matrix are passed in as arguments by reference (and consequently for further recursive calls as well).

We found it challenging to identify significant improvements to the conventional matrix multiplication algorithm in terms of reducing its asymptotic complexity. However, there is a well-known practical way to speed up the conventional matrix multiplication in a computer program, which is to loop over the matrices in a way that exploits locality of reference better. For matrices of sufficient size that span over multiple cache lines, we can reduce the number of times the CPU has to load new data into cache by ensuring the the way matrix entries are accessed are in the order in which they are present in memory. Please refer to our implementation in the file `strassen.cpp` for the specific loop order.

We now proceed to describe a specific optimization to our implementation of Strassen's algorithm for matrix multiplication. The question is how to handle matrices which have dimensions that are not a power of two? The key insight is that it is unnecessary to pad the matrix to the next largest power of two dimension[1]. Rather, if we encounter a matrix dimension which is not a power of two, we pad it to a dimension which can be cleanly divided by two until it reaches the specified crossover point. For example, given a 1599×1599 sized matrix and a specified crossover point of 25, it suffices to pad one extra row and column only since we can reach the crossover point cleanly from there, after which the conventional matrix multiplication algorithm is used. An alternative to this padding strategy is the "lazy" approach, where a matrix with odd dimension is padded on-demand with a single row and column to make the dimensions even before carrying out each divide and combine step. We elected not to use the alternative padding strategy, as we hypothesize that the computational cost of doing a one-off larger 2D-array resize at the start of the computation outweighs the cost of potentially executing multiple smaller 2D-array resizes during the recursive calls. Due to the time constraints of this assignment, we were unable to compare between the two padding strategies, but it would be an interesting area for further investigation.

[1]Higham, Nicholas J. (1990). "Exploiting fast matrix multiplication within the level 3 BLAS" (PDF). ACM Transactions on Mathematical Software. 16 (4): 352–368.

Results and Analysis

Having discussed the various implementation considerations and optimizations in the previous subsection, we now present the results of our experiment to determine the optimal crossover point n_0. To find the optimal crossover point, we tried matrices with uniformly random entry values between 0 and 65536 for matrix dimensions $n = 2, \ldots, 499$. For each dimension, we systematically try crossover points from $n_0 = 2, \ldots, \min(n, 100)$. We chose a limit of 100 conservatively based on our analytical results estimating the crossover point to be less than 50. For each crossover point trialed, we averaged the running time of the conventional matrix multiplication algorithm and Strassen's algorithm over five iterations each to minimize the variance in running time results due to external factors (e.g. other concurrent programs running on our computer during the execution of our algorithm). We reported the first crossover point that allowed Strassen's algorithm to outperform the conventional algorithm for each matrix dimension trialed. Our result table is nearly 500 rows long, so we have not included it in this report, but is available for viewing within the sub-directory (`output/strassen.xlsx`) of our submitted repository for this assignment.

Rather than the tabular results from our experiment (for reasons described above), we will present and analyze them using the following graphs below. Figure 2 shows the difference in running times of the conventional algorithm against Strassen's algorithm as the dimensions of the input matrices increase. It is observed that the conventional algorithm begins faster (or takes time at most) compared to Strassen's algorithm for smaller dimensions, and for all crossover points, until $n \approx 100$, after which Strassen's algorithm seems to gradually outperform the conventional algorithm and have a lower running time. From inspecting the actual value of our result table, we see that the first crossover point at which Strassen's algorithm outperforms the conventional algorithm is at $n = 114$ and a crossover point of $n_0 = 59$. Furthermore, the rate at which the difference grows between the two algorithms appears to increase with n. The fact that we can observe Strassen's algorithm outperforming the conventional matrix multiplication algorithm as the dimensions of the matrix grow large is a positive sign for the correctness of our implementation, since in theory we expect Strassen's algorithm to outperform the conventional algorithm asymptotically.

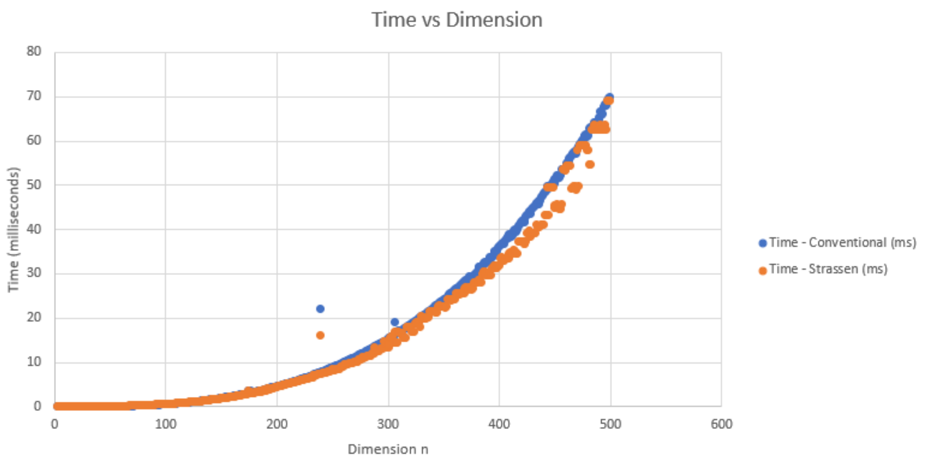

Figure 2: Comparing the expected number of triangles against the actual number counted for a range of probabilities.

To determine our optimal crossover point n_0, we present Figure 3, which displays the crossover point for each dimension of the matrix. As discussed in the preceding paragraph, the conventional algorithm outperformed Strassen's algorithm for all crossover points trialed until $n = 114$, so we do not report any crossover points for $n < 114$. We observe that the crossover points consistently fall within a range of

$n_0 = 28, \ldots, 99$ as the matrix dimensions grow large (the exact values were read from our table of results). An interesting observation is that the optimal crossover point appears to decrease as the matrix dimension grew. The crossover point for matrix dimensions between 100 and 200 are greater than 50, but for matrix dimensions between 400 and 500, the crossover point was around the 35 ± 5 mark. From these results, it appears that the asymptotic nature of the crossover point seems quite similar to our analytical results, which gave a crossover point of $n_0 = 38$ for odd-dimension matrices. We speculate that with further experiments for matrices of larger dimensions, we might even observe an eventual crossover point closer to the $n_0 = 16$ mark, as given by our analytical analysis for even-dimension matrices.

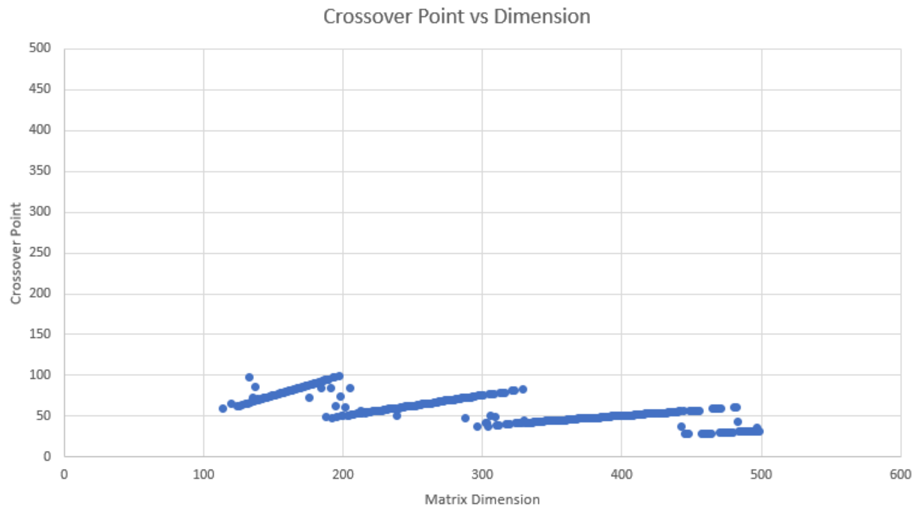

Figure 3: Comparing the expected number of triangles against the actual number counted for a range of probabilities.

As a side experiment, rather than using random values between $0, \ldots, 65536$ for our input matrix entries, we repeated the experiment with random values of 0 and 1 instead. The results of this are stored in the file `output/crossover_01.txt`. The first instance of Strassen's algorithm outperforming the conventional algorithm was lower at $n = 87$, and a crossover point of $n_0 = 50$, though the timings of each algorithm remained largely the same when compared to using random values between $0, \ldots, 65536$. This suggests that the lower crossover point value may have been due to variance in running times of trials, rather than anything of interest.

To conclude this section, we take the average over all the crossover points found for the dimensions where Strassen's algorithm outperforms the conventional algorithm, which we suggest as our optimal crossover point. This value is $n_0 = 57$ (out of interest, the median crossover point was 55). We speculate that this crossover point could be lowered with more trials on matrices with larger dimensions, perhaps an interesting topic for further investigation in the future. However, we note that it would be unrealistic for our implementation to match the results of the analytical analysis, since we assumed that the cost of all operations other than integer arithmetic were free. Although useful theoretically since it greatly simplifies analysis, this is not practical when measuring running times in reality, as there is a real (time and power) cost to executing instructions other than integer arithmetic on a computer. Allocating and deallocating memory to represent large matrices in our program are one such example of an expensive operation not considered by our analytical analysis. Despite this, it is clear from the results of our experiment that Strassen's algorithm is useful in practice since modern methods in scientific computing and artificial intelligence definitely use matrices with dimensions greater than 100×100.

Counting Triangles in Random Graphs

As an aside to our investigation into the optimal crossover point for Strassen's divide and combine matrix multiplication algorithm, our implementation of Strassen's algorithm allows us to experiment with finding triangles in random graphs. This section describes the results of counting the number of length-3 cycles (triangles) in a random graph, and how this changes with the probability p that an edge is present in the graph. These two ideas are linked since the entry indexed by row i and column j of the adjacency matrix raised to the n^{th} power gives us the number of length n paths from vertex i to vertex j. We therefore compute the third power of the adjacency matrix and sum the values in the diagonal. It is important to remember that summing the entries of the diagonal in the straightforward way over counts the number of triangles by a factor of six, since each vertex of a triangle is counted twice in either direction of the cycle.

To carry out this experiment, we fix the value of p to be one of the values in the set of probabilities $\{0.01, 0.02, 0.03, 0.04, 0.05\}$, and compare the output using our algorithm for Strassen's matrix multiplication to the empirical formula for the expected number of triangles, given by $\binom{1024}{3}p^3$. Since we use a pseudorandom number generator in our program to determine whether an edge exists, we take the average over fifteen trials to minimize the influence of any one trial. From Table 2 and Figure 4 shown below, the difference between the expected number of triangles and the actual number of triangles counted by our program lie between 0.11% and 1.06%. In fact, the results were so similar that it is somewhat difficult to see the two sets of values in the displayed graph. We can thus conclude that our implementation of Strassen's algorithm appears to be accurate.

p	Expected (empirical)	Actual Observed	Absolute Difference (%)
0.01	178.43	178.20	0.13
0.02	1427.46	1412.50	1.06
0.03	4817.69	4823.15	0.11
0.04	11419.71	11464.00	0.39
0.05	22304.13	22400.30	0.43

Table 2: Comparing the differences between the empirical formula and the results from our implementation.

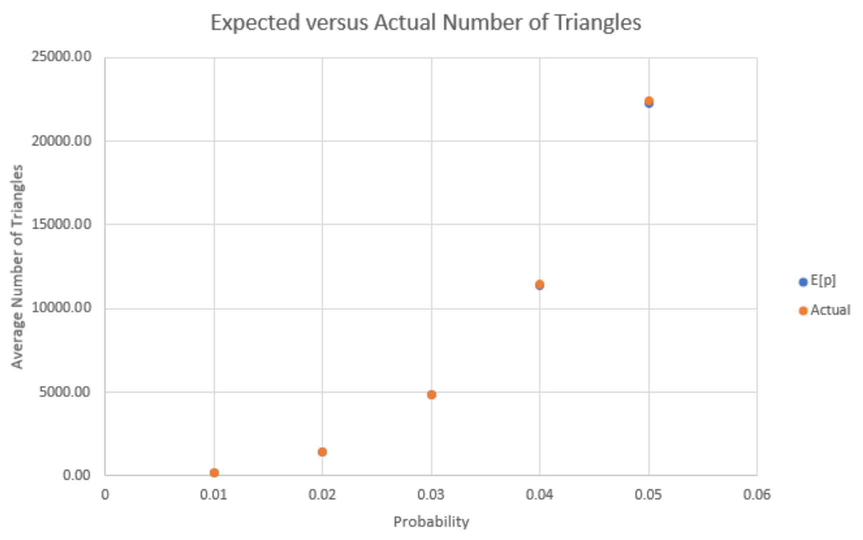

Figure 4: Comparing the expected number of triangles against the actual number counted for a range of probabilities.

Conclusion

In summary, we have used two different methods to identify the optimal crossover point n_0 for Strassen's matrix multiplication algorithm. Firstly, the analytical analysis carried out suggests a value of $n_0 = 16$ for even-dimension matrices, and $n_0 = 38$ for odd-dimension matrices. This was confirmed using a simple numerical analysis using those crossover points to simulate the number of operations using each matrix multiplication algorithm.

On the other hand, our experimental analysis suggested a crossover point of $n_0 = 57$. This was from using a brute-force approach to try the entire range of matrix dimensions until $n = 500$, with increasing crossover points to check if there was a crossover point which would allow Strassen's matrix multiplication algorithm to outperform the conventional algorithm. As discussed in the previous sections, we believe that it is possible to further reduce the crossover point found by Strassen's algorithm by averaging in the results of matrices with higher dimensions, but that it would not be possible to do as well as the analytical analysis. The main reason for this lower bound is that the analytical model considered all other operations outside of matrix multiplication to be free.

In this experiment, we also investigated the behavior of triangles in random graphs since our implementation of Strassen's algorithm provided an efficient way to multiply matrices. We found that our results correlated well with the expected number of triangles over all the specified probabilities $0.01, 0.02, 0.03, 0.04, 0.05$, calculated using empirical means. This reinforced confidence in our implementation of Strassen's matrix multiplication algorithm.

Taking the results of both our analytical and experimental analysis together, it is clear that Strassen's algorithm does have a place in the modern computing world. The textbook Algorithms in C (1990 edition) was perhaps somewhat over-exuberant in downplaying the practicality of Strassen's algorithm, since our analytical and experimental analysis have shown that it offers a speed advantage over the traditional algorithm over relatively smaller values of n. The claim that n would have to be in the thousands before Strassen's algorithm becomes useful does not appear to agree with the results of the experiment we have carried out. Our experiments have shown that Strassen's algorithm offers an advantage once we reach dimensions of $n \approx 100$. Furthermore, in the problems encountered in the modern day in scientific computing and artificial intelligence use matrices with dimensions that are significantly larger than this, and so the benefits of using Strassen's algorithm grows asymptotically too!

That being said, it is difficult not to discount the effect that computing hardware has on executing programs in the real world. The actual time and power cost of running various operations can differ greatly depending on the computer architecture (for example, the same instruction can take different number of CPU clock cycles on different architectures). Therefore, it is understandable that the experimental analysis carried out on computers in 2023 may have had wildly differing results to say computers in 1990.

We now provide some scope to further develop the results from this experiment in the future. Firstly, we would be interested to know if there are more efficient padding strategies than the one used in our implementation. Specifically, a comparison with the "lazy" padding strategy described in the preceding sections. Secondly, although we believe it is possible to lower the suggested crossover point from the experimental analysis through carrying out further trials on matrices of higher dimensions, this has not been shown from our results. Therefore, we would be interested to know if running the experiment further on larger sized matrices would further reduce the averaged crossover point (and what would be the lower bound of this?). Lastly, it would be interesting to compare the crossover points generated by different computer architectures, not just in the modern day but computers in the 1990. This might provide some context into the aforementioned claim that lies at the heart of our experiment.