Programming Assignment 2

Jennifer Guo and Ayana Yaegashi

March 30, 2022

1 Introduction

In this report we will discuss the results of our experiments as we determine the optimal crossover point for Strassen's matrix multiplication algorithm to switch into a conventional matrix multiplication algorithm. We will compare the results of our experimentally determined crossover point for matrices of varying dimensions to our mathematically determined optimal crossover point. In addition, we will discuss how we used our implementation of Strassen's algorithm to count the number of triangles in random graphs. We used C++ to implement both Strassen's algorithm and a conventional matrix multiplication algorithm, as well as to run our experiments.

1.1 Mathematically Determining the Crossover Point

The conventional matrix multiplication algorithm is an $O(n^3)$ algorithm. Strassen's algorithm makes an improvement on this time complexity, using recursion to calculate 7 subproblems of size $\frac{n}{2}$ and spending $O(n^2)$ time to combine the subproblems. Therefore, Strassen's algorithm gives a $O(n^{2.81})$ algorithm by the Master Theorem. It's clear that asymptotically, Strassen's algorithm does better with matrices of large dimensions. However, there becomes a point where the conventional matrix multiplication algorithm is faster for smaller dimensions. We will mathematically determine the crossover point n_0 in which running the conventional matrix multiplication algorithm on an $n_0 \times n_0$ matrix is just as fast if not faster than how Strassen's would run on an $n_0 \times n_0$ matrix. Then, for all $n \leq n_0$, Strassen's will switch over to the conventional algorithm to optimize overall runtime.

To mathematically determine n_0, we examined recurrences for Strassen's algorithm and a conventional algorithm. We obtained an optimal n_0 for $n \times n$ matrices where the dimension n is even as well as an optimal n_0 for $n \times n$ matrices where the dimension n is odd.

Strassen's algorithm uses a divide and combine approach to recursively conduct 7 multiplications of subproblems of size $\frac{n}{2}$. In the "combine" step, Strassen's then must make 18 matrix additions/subtractions of $\frac{n}{2} \times \frac{n}{2}$ sized matrices, each with $(\frac{n}{2})^2$ elements. The additions and subtractions take $O(1)$ time, so combining subproblems takes $18(\frac{n}{2})^2$ time. Overall, we can model the runtime of Strassen's with the following recurrence.

$$S(n) = 7S(\frac{n}{2}) + 18(\frac{n}{2})^2$$

On the other hand, the conventional matrix multiplication algorithm makes n multiplications and $n-1$ additions (a total of $2n-1$ operations for each of the n^2 elements). Therefore, the runtime of the conventional algorithm can be expressed as follows.

$$T(n) = n^2(2n-1)$$

We are solving for n_0, which is the dimension in which Strassen's and the conventional algorithm have the same runtime. Therefore we are looking for the value of n_0 such that

$$S(n_0) = T(n_0)$$

In addition, at n_0, Strassen's will switch over to using the conventional algorithm, so we can relate the equations as follows.

$$S(n_0) = 7 \cdot T(\frac{n_0}{2}) + 18 \cdot (\frac{n_0}{2})^2 = T(n_0)$$

Plugging in the conventional algorithm's runtime, we have the following equation we can use to solve for n_0.

$$7 \cdot (2 \cdot (\frac{n_0}{2})^3 - (\frac{n_0}{2})^2) + 18 \cdot (\frac{n_0}{2})^2 = 2n_0^3 - n_0^2$$

Algebraically solving for n_0, we determine n_0 for matrices where dimension n is even.

$$n_0 = 15, \text{ n is even}$$

For values of n that are odd, we simply pad our matrix with one row and one column of zeroes (explanation in the next section) which turns the matrix into an n+1 matrix. Therefore, our recurrence for Strassen's with odd n is as follows.

$$S(n) = 7S(\frac{n+1}{2}) + 18(\frac{n+1}{2})^2$$

Using the same method as previously to solve for n_0, we have that:

$$7 \cdot (2 \cdot (\frac{n_0+1}{2})^3 - (\frac{n_0+1}{2})^2) + 18 \cdot (\frac{n_0+1}{2})^2 = 2n_0^3 - n_0^2$$

Solving for n_0, we determine n_0 for matrices where n is odd.

$$n_0 = 26.867, \text{ n is odd}$$

2 Implementation

2.1 Representing Matrices

We decided to create a `Matrix` struct to represent matrices in our implementation. `Matrix` has 4 properties: `values` which is a pointer to a 2-dimensional integer array of elements, `dimension` which stores an integer representing the dimension n of the matrix, `startRow` which stores an integer representing the index of the first row of elements to look at in `values`, and `startColumn` which analogously stores an integer representing the index of the first column of elements to look at in `values`.

Our implementation of the `Matrix` struct was an important optimization in our implementation of Strassen's algorithm. In the step where Strassen's algorithm partitions the original matrix of dimension n into 4 equal parts each of dimension $\frac{n}{2}$ (matrices $A, B, ..., H$) we realized it would be inefficient to allocate space for four more 2-dimensional integer arrays of values. Instead, since the 4 smaller matrices are each a portion of the original matrix, we simply create four new `Matrix` objects in which each `values` property points to the same `values` array in the original matrix. Then, we only have to halve the `dimension` and shift the `startRow` and `startColumn` values to keep track of the top-leftmost element in each of the four smaller matrices accordingly. Thus, this optimization allowed us to avoid 8 memory allocation and memory freeing operations per call to Strassen's. In addition, the optimization also allows us to avoid unnecessary copying of matrix values.

2.2 Implementing Strassen's and Conventional Matrix Multiplication

Implementing Strassen's algorithm was straightforward for matrices of dimension n where n is a power of 2 because we could easily split the matrix into four sub-matrices at each step. In order to work with matrices of dimension n where n is not a power of 2, however, we had to safeguard the fact that the matrices could have an odd n in which case we could not easily divide them into 4 equally-sized smaller matrices.

One solution would be to simply pad the matrix with enough rows and columns of 0s to round n to the next highest power of 2, but we realized this would be an inefficient solution. For example, if we were given matrices with dimension 513, we would have to pad the matrices to dimension 1024, almost doubling the size of the problem. Instead, we decided to only pad the matrix whenever necessary. In other words, at each recursive step we would take a look at the size of the subproblem n (the dimension of the matrix). If n was odd, we would pad the matrix with one row and one column of 0s so that n could be evenly split into sub-matrices. If n was even, we would leave it alone. This way, we only increased the size of the problem by 1 whenever necessary, a great improvement from the padding to the next power of 2 solution.

We also implemented the conventional matrix multiplication algorithm. After testing to make sure our conventional algorithm was correct, we wrote a function `checkCorrectness` that would compare the output of Strassen's algorithm to the output of the conventional algorithm on the same matrices so we could easily determine the correctness of our implementation of Strassen's. In addition, we wrote function `generateRandomMatrix` to easily generate random matrices with elements $\{-1, 0, 1\}$ or $\{0, 1, 2\}$ of varying dimensions to test on.

2.3 Optimizing Algorithm Efficiency

As described in section 2.1, we never had to allocate new memory when splitting the original matrices into 4 smaller matrices each of size $\frac{n}{2} \times \frac{n}{2}$ (matrices $A, B, ..., H$) because instead we used the properties of our `Matrix` struct to simulate creating 8 half-dimension matrices by keeping track of the `startRow` and `startColumn` values and setting `dimension` $= \frac{n}{2}$. Another optimization was, as described in section 2.2, to only pad the matrix with one row and one column of zeroes whenever necessary, instead of padding the matrix to the next power of 2. This prevented us from having to potentially almost double the dimension of our matrix, and thus made the algorithm more efficient.

However, while testing our implementation we noticed that during each iteration of Strassen's, we were dynamically allocating memory each time we added two matrices together, which was necessary to produce matrices $P1, ..., P7$. Instead of allocating and freeing memory for 18 intermediate matrices, we realized we only had to allocate memory for 2 temporary matrices. We can continuously store the values of the sum of two matrices in these temporary matrices, since after we use the temporary matrix to run Strassen's again on the subproblems, we didn't need them anymore. We needed 2 temporary matrices instead of 1 because there were some subproblems in Strassen's, such as calculating $P_5 = (A + D)(E + H)$ where we had to store the value of two sums instead of just one.

In addition, after calculating $P1, ..., P7$, we realized we didn't have to allocate memory for the matrices representing the four corners of the resulting matrix ($AE + BG$, $AF + BH$, $CE + DG$, and $CF + DH$). Instead, we simply allocated memory for our resulting matrix `Product` of dimension n, and manipulated the `dimension`, `startRow`, and `startColumn` properties of the matrix struct to directly store the sums representing the four corners into the `Product` matrix. With these two optimizations in which we reused memory, we were able to drastically decrease the number of matrices allocated at each iteration of Strassen's, which greatly optimized our runtime.

2.4 Experimentally Finding the Crossover Point

For our experimentation, we decided to test values of potential crossover points from $n_0 = 15$ (our theoretical crossing point for even n) to $n_0 = 195$ for even n, and from $n_0 = 35$ to $n_0 = 245$ for odd n.

While we did not expect that our experimental crossing point would actually be so high as 195 or 245, we want to test enough values of n_0 such that we could be certain that we actually found the right value of n_0 that minimized running time. If our upper bound for n_0 were lower, we may have run the risk of mistaking fluctuations and noise in the data for an upward trend in running time after hitting the minimum. As a result, then there would have been danger of underestimating our n_0 values.

For our even n values, we decided to test $n = 1000, 1100, 1200, 1300, 1400$. For our odd n values, we decided to test $n = 1005, 1105, 1205, 1305, 1405$. We chose to start at $n = 1000$ because we wanted to make sure our n was large enough that the results of our trials would not be severely affected by noise. Initially, we did some trial runs where $n = 400, 600$, and 800. In these trials, the experimental runtimes were much closer together and it was more difficult to discern the minimum runtime from just fluctuations in runtime as a result of noise. For larger values of n (which we determined to be $n \geq 1000$) the asymptotic efficiency of Strassen's algorithm becomes more clear because the operations of adding, subtracting, and so on (operations that were accounted for in the theoretical running time of Strassen's algorithm) begin to outweigh the noisiness introduced by operations related to memory allocation, for instance. Thus, we thought we would get the clearest data analyzing large values of n.

3 Results

3.1 Crossover Point

The following tables present the results of our trials to find the experimental value of n_0 for our implementation of Strassen's algorithm. For each value of n, we ran 10 trials per n_0 value and recorded the amount of time in milliseconds that it took to calculate the matrix product. The numbers below are the average times in milliseconds over 10 trials to calculate the matrix product for different combinations of n and n_0. We split our results into individual tables for n_0 values from testing even and odd n. Recall from section 1 that for even n we calculated n_0 to be about 15 and for odd n we calculated n_0 to be almost double at 26.867. We expect a higher n_0 value for odd n as a result of having to pad larger matrices upfront.

n_0	$n = 1000$	$n = 1100$	$n = 1200$	$n = 1300$	$n = 1400$
15	1703.159	2448.574	3126.62	3633.95	3433.03
25	841.520	1192.3	1381.74	1761.28	1770.14
35	566.099	787.715	1511.64	1615.51	1739.59
45	571.788	841.931	973.083	1148.32	1284.36
55	575.849	827.489	966.259	1075.72	1194.64
65	431.373	827.609	878.443	957.866	1219.17
75	446.750	623.057	724.375	932.73	1324.67
85	422.322	630.591	729.583	881.6	1292.42
95	429.684	623.796	776.424	843.181	1323.91
105	435.197	629.701	876.455	863.994	1348.85
115	422.233	627.609	776.735	857.1	1292.48
125	457.091	633.315	772.944	876.601	1352.6
135	453.872	661.94	820.673	932.822	1286.84
145	468.858	731.265	753.859	850.756	1304.6
155	451.299	645.517	936.274	873.672	1258.71
165	451.282	644.107	928.264	964.252	1286.95
175	456.829	643.137	932.679	926.861	1313.21
185	454.301	656.51	875.561	954.713	1315.52
195	453.879	669.52	1017.3	950.066	1357.55

Table 1: Our experimental results show that for even n, n_0 is about 75, and ranges from 55 to 95.

n_0	$n = 1005$	$n = 1105$	$n = 1205$	$n = 1305$	$n = 1405$
35	624.553	948.171	1059.45	1514.15	1512.53
45	540.755	986.888	817.923	948.236	1085.16
55	596.126	932.204	775.672	949.661	1074.69
65	429.365	740.503	812.892	925.197	1251.43
75	419.108	720.76	779.885	948.195	1260.11
85	418.702	731.27	701.857	850.516	1131.93
95	423.96	799.801	693.643	912.958	1112.92
105	416.68	766.576	702.701	872.757	1316.67
115	426.15	750.007	702.541	869.731	1049.76
125	435.456	1001.19	714.646	873.26	1060.38
135	500.018	770.041	852.316	836.098	1112.28
145	478.68	777.838	690.9	868.932	1108.55
155	616.456	8459.32	744.411	893.875	1144.38
165	528.875	2349.78	760.797	1039.39	1228.68
175	502.597	699.877	748.31	1342.14	1085.53
185	495.194	716.626	778.272	972.571	1301.13
195	502.515	833.038	760.673	1046.53	1366.34
205	526.88	750.926	816.257	984.704	1230.33
215	508.162	752.026	763.954	1006.73	1262.95
225	499.185	789.574	841.894	1100.81	1237.28
235	548.762	818.771	756.604	1161.68	1277.31
245	506.177	758.527	801.699	1004.84	1277.05

Table 2: Our experimental results show that for odd n values n_0 ranges from 105 to 175.

3.2 Counting Triangles in Random Graphs

Probability p	**Expected No. of Triangles**	**Avg. Actual No. of Triangles**
0.1	178	178
0.2	1427	1424
0.3	4818	4824
0.4	11420	11397
0.5	22304	22406

Table 3: Table of the average number of triangles over 10 trials experimentally appearing in random graphs with 1024 vertices and probability p of an edge.

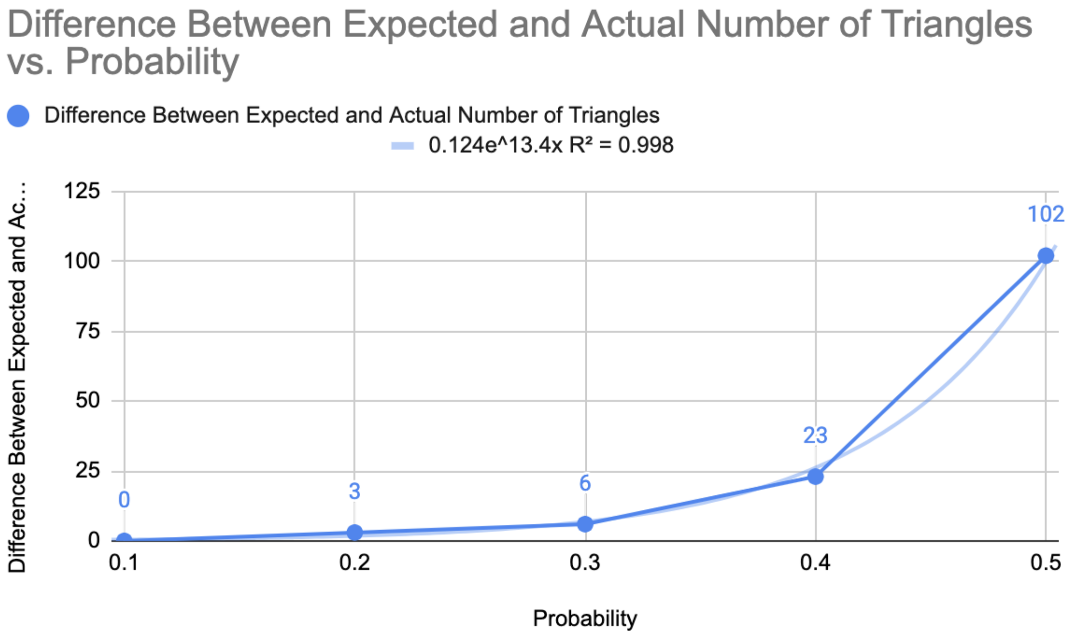

Figure 1: Graph detailing the difference between the expected and actual number of triangles at each probability p.

4 Discussion

4.1 Crossover Point

In section 1.1, we determined theoretically that $n_0 = 15$ if n is even and $n_0 = 26.867$ if n is odd. As displayed in section 3.1, our experimentally determined n_0 ranged between 55 and 95 (and was most frequently 75) when n is even and between 105 and 175 when n is odd. Notice that similarly to our theoretical calculations, n_0 for odd n values is roughly twice as large as n_0 for even n numbers. However, we also observed that our experimental n_0 values were larger than our theoretically calculated n_0 values. We believe there are several reasons for this discrepancy between our theoretically and experimentally determined n_0 values.

First, our analytically determined n_0 values do not take into consideration the time to allocate and free memory, which was necessary in our implementation of Strassen's algorithm since we had to deal with very large matrices. Although we made a series of optimizations, detailed in section 2.3 above, we still needed to allocate memory for the following matrices in our implementation:

- Matrices of size $n + 1 \times n + 1$ when we padded an odd-length matrix.
- `temp1` and `temp2` matrices each of size $\frac{n}{2} \times \frac{n}{2}$ in order to store intermediary matrix addition results for matrices $P1$ through $P7$.
- One matrix of size n (or $n + 1$ depending on padding) to store the calculated product matrix, which the algorithm returns.

These allocations, which are not taken into consideration in our theoretical analysis, add runtime to our implementation of Strassen's algorithm. Operations related to memory management favor the conventional matrix multiplication algorithm. In the conventional algorithm, there is one upfront cost of allocating $n \times n$ space in memory for the resultant matrix. On the other hand, in our implementation of Strassen's algorithm, the higher that n_0 is, the fewer recursions Strassen's algorithm does and the fewer the number of memory mannagement operations are done. Thus, for a while beyond the theoretically calculated n_0 values, it is actually practically optimal to switch to the conventional matrix multiplication algorithm earlier. This is also a reason why we felt that it was important to try to optimize our memory allocation strategies as best as we could in order to avoid unnecessary allocation and de-allocation operations.

In addition, when theoretically finding n_0, we made many assumptions about the cost of operations. We assumed that the cost of any single arithmetic operation (adding, subtracting, multiplying, or dividing two real numbers) is constant $O(1)$ time. However, this is not the case in practice, which is another reason why our experimental n_0 is higher than expected.

As can be seen in our results table, the experimental value of n_0 is not consistent within our results. For instance, n_0 when $n = 1300$ is larger than n_0 when $n = 1400$. One reason for this is possibly because of the values of n themselves. Take for example the difference between $n = 1400$ and $n = 1300$. 1400 divides by two 3 times before we reach an odd number. Thus, the first time that our algorithm would have to resize the matrix and pad it would be when $n = 1400/2/2/2 = 175$. On the other hand, 1300 divides only 2 times before we arrive at an odd number: 325. Thus, when working with 1300, the algorithm must allocate memory for a much larger matrix to pad. Thus, it would be reasonable to conclude that our algorithm is generally slower for $n = 1300$ than for $n = 1400$, and so crossing over to the conventional matrix multiplication algorithm at a higher n_0 makes more sense.

For $n = 1100$, we noticed that there are two n_0 for which the time taken to calculate the matrix product was at a nearly identical 623 milliseconds. Similarly to what is discussed in the paragraph above, we may have seen a dip in running times at these two values of n_0 because they lent themselves to a particularly smooth series of divisions by 2, which allowed the algorithm to avoid allocating memory for padding purposes. We decided to highlight the first minimum n_0 value, first because it is the true minimum and second because we felt that $n_0 = 75$ was more consistent with the results we found from our trials of other n. A similar phenomenon occurs with $n = 1000$ where both $n_0 = 85$ and $n_0 = 115$ produce roughly the same experimental runtime. We believe this likely occurred for a similar reason to that just discussed, and we ultimately decided to go with $n_0 = 85$ since it was also most consistent with the results produced by our other n.

Regarding our data for odd n, we noticed first that the n_0 values we produced were higher than those produced for even n, and second that the range of n_0 values was more spread out. With regard to the first observation, we believe that this is because the memory allocation operations needed for padding such large matrices makes Strassen's less efficient. For instance, for an even n such as $n = 1000$, the first time that we need to pad a matrix is when we have recursed down to $n = 125$. On the other hand, for any odd n such as $n = 1005$ we immediately have to pad the matrix and allocate memory for a matrix of dimension 1006. Thus, this general pattern explains why larger experimental runtimes are reasonable for odd n and are also consistent with the theoretical calculations we did in section 1.

With regard to our second observation that the data was more spread out with our odd n values, one reason for this may be with regard to how the number divides. Consider $n = 1005$ versus $n = 1105$, for which there is a large gap in n_0 values. For $n = 1005$, the algorithm needs to pad the initial matrix and then since $1006/2 = 503$, the smaller matrices of $n = 503$ also needed to be padded. However, after this, all subsequent n divide cleanly into even numbers so the next time the algorithm will need to pad is for a matrix of size $n = 63$. On the other hand, when $n = 1105$ the algorithm must pad each half-sized matrix all the way until $n = 70$. There is clearly much more padding required for $n = 1105$ than for $n = 1005$, and thus we believe this explains why n_0 is so much larger for $n = 1105$ than for $n = 1005$. Analyzing how matrices for which $n = 1205, 1305$, and 1405 we found that while there were more padding operations required than for $n = 1005$, there were fewer than for $n = 1105$, explaining why their n_0 values fall generally in between.

4.2 Counting Triangles in Random Graphs

We also used our implementation of Strassen's algorithm to count the number of triangles in random graphs represented by a 1024×1024 adjacency matrix, where each edge had probability $p = 0.1, 0.2, 0.3, 0.4, 0.5$ of showing up. Table 3 illustrates the average actual number of triangles counted on random graphs over 10 trials for each probability p, as well as the expected number of triangles given by $\binom{1024}{3}p^3$. Something to notice is that the average actual number of triangles for each probability p is very close the expected number of triangles. Having conducted 10 trials each, we believe this result can be attributed to the law of large numbers, as when we conduct more trials on random graphs, the average result should grow closer to the true expected value.

Additionally, as seen by Figure 1, the difference between the expected and actual number of triangles counted increases exponentially as the probability increases. We believe this makes sense because as the probability p increases, there are more edges in the resulting graph. Therefore, there are more non-zero entries in the adjacency matrix A and when A^3 is calculated and the triangles are counted, there will be greater variability in the number of triangles found in each random graph.