CS124 Programming Assignment 2

Anusha Murali and Alejandra Perea Rojas

March 27, 2024

1 Introduction

Consider square matrices $A, B \in \mathbb{R}^{n \times n}$ and their product $C = AB$. For ease of analysis, let n be a power of two. Let us express these matrices as made up of block matrices of size $n/2 \times n/2$ as following:

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}, \quad C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}.$$

Since $C = AB$, we have the following:

$$\begin{aligned} c_{11} &= a_{11}b_{11} + a_{12}b_{21} \\ c_{12} &= a_{11}b_{12} + a_{12}b_{22} \\ c_{21} &= a_{21}b_{11} + a_{22}b_{21} \\ c_{22} &= a_{21}b_{12} + a_{22}b_{22} \end{aligned}$$

As we see from the computation of matrix C above, using the conventional matrix multiplication method, we require 8 matrix multiplications between matrices of size $n/2 \times n/2$ and 4 matrix additions.

Strassen's algorithm cleverly reduces the number of multiplications from 8 to 7, thereby improving the matrix multiplication when the size of the matrices is large. We note that the number of additions increases to 18 from 4, however the smaller number of multiplications benefits more than the work required for the extra additions. Specifically, Strassen's algorithm computes the following variables:

$$\begin{aligned} m_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\ m_2 &= (a_{21} + a_{22})b_{11} \\ m_3 &= a_{11}(b_{12} - b_{22}) \\ m_4 &= a_{22}(b_{21} - b_{11}) \\ m_5 &= (a_{11} + a_{12})b_{22} \\ m_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\ m_7 &= (a_{12} - a_{22})(b_{21} + b_{22}) \end{aligned}$$

and uses them to compute c_{11}, c_{12}, c_{21} and c_{22} as follows:

$$\begin{aligned} c_{11} &= m_1 + m_4 - m_5 + m_7 \\ c_{12} &= m_3 + m_5 \\ c_{21} &= m_2 + m_4 \\ c_{22} &= m_1 - m_2 + m_3 + m_6 \end{aligned}$$

Therefore, we can write the following recurrence equation to represent the runtime of Strassen's algorithm for matrix multiplication:

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

On the other hand, the conventional matrix multiplication requires n^3 multiplications (as we need to find the dot product of each row and each column) and $n^2(n-1)$ additions. Therefore, the conventional matrix multiplication has the following runtime:

$$T(n) = n^3 + n^2(n-1) = 2n^3 - n^2.$$

During Strassen's algorithm, as we halve the size of the matrix each time, we will reach a crossover point where switching over to the conventional algorithm would result in lower cost. Let us say that the crossover happens at $n = n_0$. Therefore, we can write the following equation:

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

Solving, we obtain $n_0 = 15$. Since we assumed n is a power of 2, we can round up the crossover value and obtain $n_0 = 16$.

Suppose that we arrive at an odd value of n_0 during the recurrence, we can include an extra row and extra column of 0's to pad the matrix to make its size even. Therefore, we can solve for the crossover point using the following equation:

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

Solving, we obtain $n_0 \approx 37.17$. Since we assumed n is an odd number, we can round up the crossover value and obtain $n_0 = 39$.

Therefore, the size of the matrix when Strassen's algorithm should switch over to the conventional matrix multiplication algorithm is,

$$n_0 = \begin{cases} 16 & \text{for even } n, \\ 39 & \text{for odd } n. \end{cases}$$

2 Implementation

In the introduction section, we found that Strassen's algorithm leads us to the recursion $T(n) = 7 \cdot T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2$, which using Master's theorem gives us a runtime of $\Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$. On the other hand, the traditional multiplication algorithm has a runtime of $\Theta(n^3)$. Even though Strassen's algorithm is asymptotically expected to run faster than the conventional multiplication algorithm, due to the large coefficient of 18 in the constant work, we expect that the conventional algorithm will outperform Strassen's for smaller n's. Therefore, we can improve the runtime of Strassen's algorithm by switching over to the conventional algorithm when the dimension of the matrices fall below a certain crossover point. Since Strassen's algorithm is a recursive algorithm, when the matrices reach the size of the threshold crossover size (n_0), we will switch over to the conventional matrix multiplication algorithm.

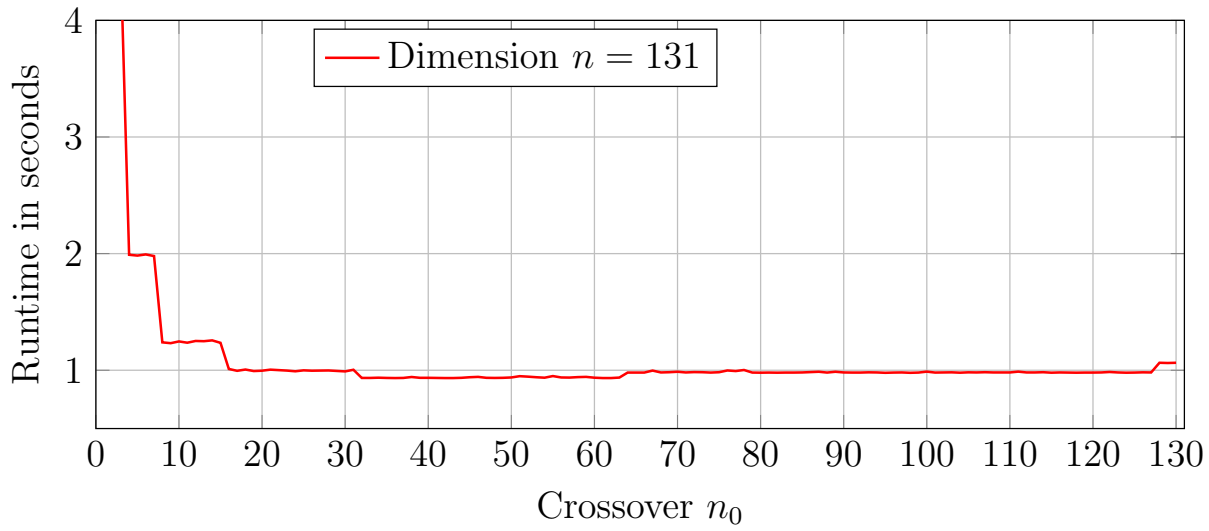
Our Python implementation of Strassen's algorithm can be used to multiply both even and odd sized matrices. We considered two different strategies to pad an odd-sized matrix. The first one was to add just one row of 0's and one column of 0's whenever we encountered an odd-sized matrix during the recursion of Strassen's algorithm. The second strategy was to identify the next larger power of two and pad the input matrices with sufficient number of columns of 0's and rows of 0's to make their sizes a power of two. The resultant matrix was obtained by only copying the entries for $i = 1, \ldots, n$ and $j = 1, \ldots, n$, from the padded result matrix. In both cases, our experiments yielded the same optimal crossover points for all matrix sizes that we considered. The runtimes too were nearly similar for both types of padding. Due to the simplicity the latter approach, where the padding is being done to the next higher power of two, we used it across all our experiments.

In order to test our algorithm, we randomly created square matrices of varying sizes from 128×128 to 2048×2048, where each entry is randomly selected to be 0, 1, or -1. We also created matrices of odd sizes such as 131×131 and 513×513. Such matrices are padded with columns and rows of zeroes so that their effective sizes will be the next larger power of two. For example, a matrix of size 131×131 will be zero-padded so that it will become a 256×256 matrix and a matrix of size 513×513 will be zero-padded so that it will become a 1024×1024 matrix. This allows us to test our implementation with both even and odd n.

3 Experimental Results and Analysis

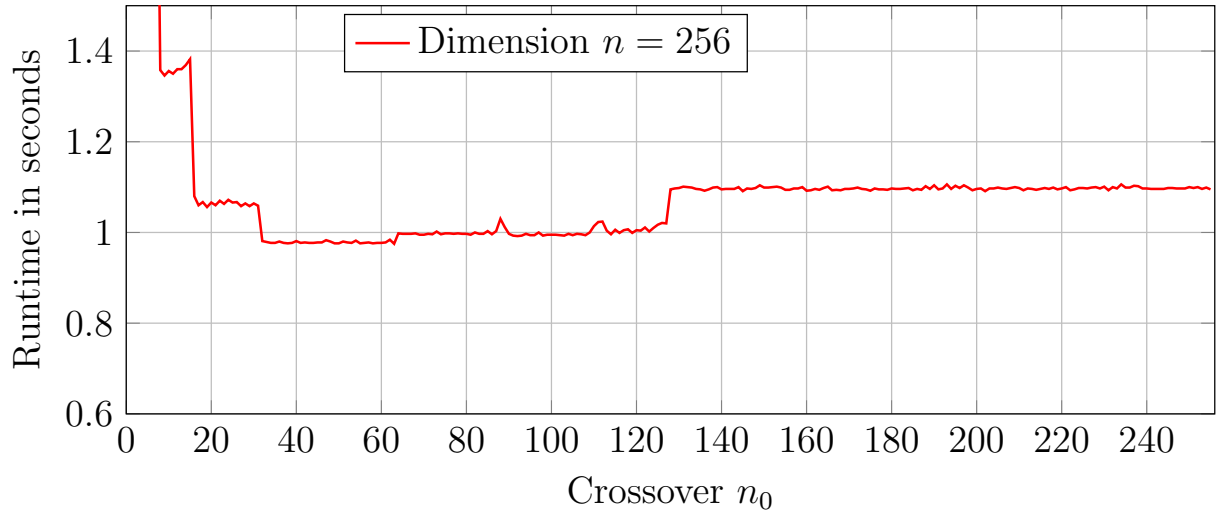
As we found in the Introduction section, when the size of the original matrices is even, then the analytical crossover size $n_0 = 16$, and when the size of the original matrices is odd, the analytical crossover size $n_0 = 39$. We conducted a number of experiments using the modified Strassen's algorithm with varying matrix sizes and varying crossover points to determine the optimal crossover points experimentally and compare them with our analytical findings. Our experimental results are presented below.

3.1 Multiplication of two 131×131 matrices



For the input size $n = 131$, we zero-pad the input matrices so that their dimensions are 256×256. The runtime of the Strassen's algorithm, without any crossover point, took 42.6 seconds (averaged over 10 runs) to multiply two 131×131 matrices, while the runtime of the conventional $O(n^3)$ matrix multiplication algorithm took only 0.16 seconds for multiplying the same dimensions. As can be seen from the plot of the experimental results above, the runtime decreased to approximately 0.86 seconds between the crossover points of $n_0 = 30$ to $n_0 = 50$, which is still higher than the runtime of the conventional algorithm.

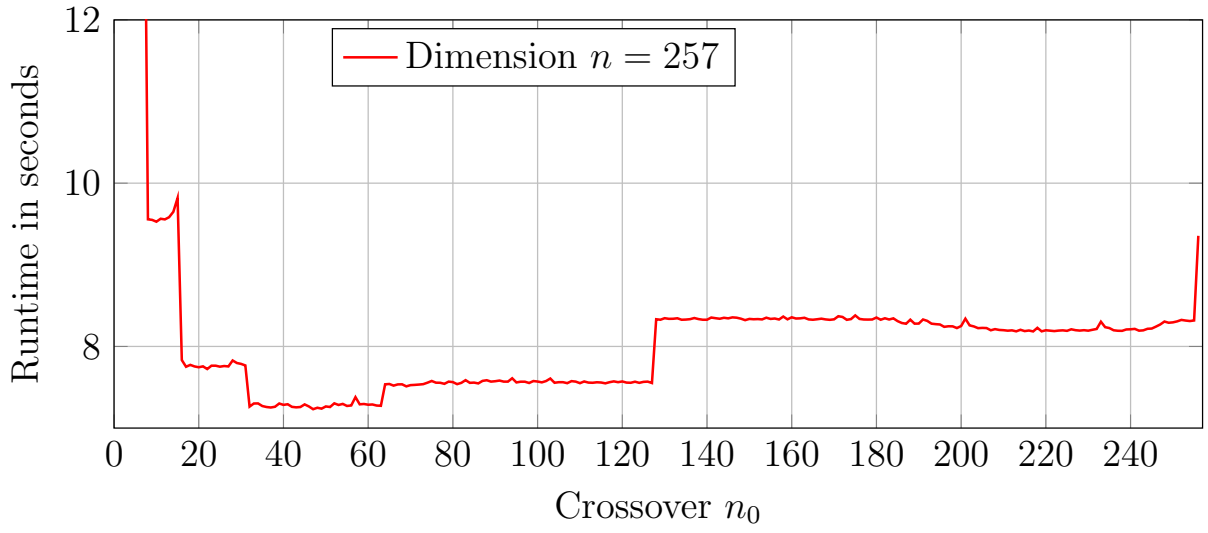
3.2 Multiplication of two 256×256 matrices



The runtime of the Strassen's algorithm, without any crossover point, took 12.8 seconds (averaged over 10 runs) to multiply two 256×256 matrices, while the runtime of the con-

ventional $O(n^3)$ matrix multiplication algorithm took only 1.11 seconds (averaged over 10 runs) for multiplying the same dimensions. As can be seen from the plot of the experimental results above, the runtime decreased to approximately 0.87 seconds between the crossover points of $n_0 = 30$ to $n_0 = 50$, which is a saving of approximately 32.6% over the conventional algorithm. This crossover range is higher than our analytical crossover point of $n_0 = 16$.

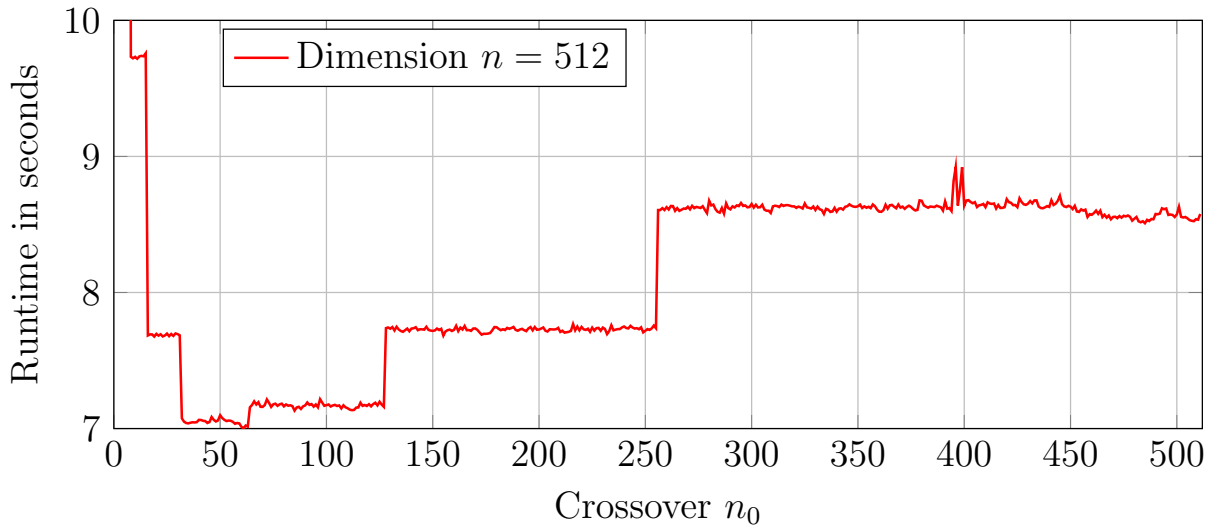
3.3 Multiplication of two 257×257 matrices



The runtime of the Strassen's algorithm, without any crossover point, took 297 seconds (averaged over 5 runs) to multiply two 257×257 matrices, while the runtime of the conventional $O(n^3)$ matrix multiplication algorithm took only 1.14 seconds (averaged over 5 runs) for multiplying the same dimensions. As can be seen from the plot of the experimental results above, the runtime decreased to approximately 7.25 seconds between the crossover points of $n_0 = 30$ to $n_0 = 55$. We note that even with the optimal crossover strategy, the runtime using the modified Strassen's algorithm is still significantly higher than the runtime of the conventional $O(n^3)$ algorithm.

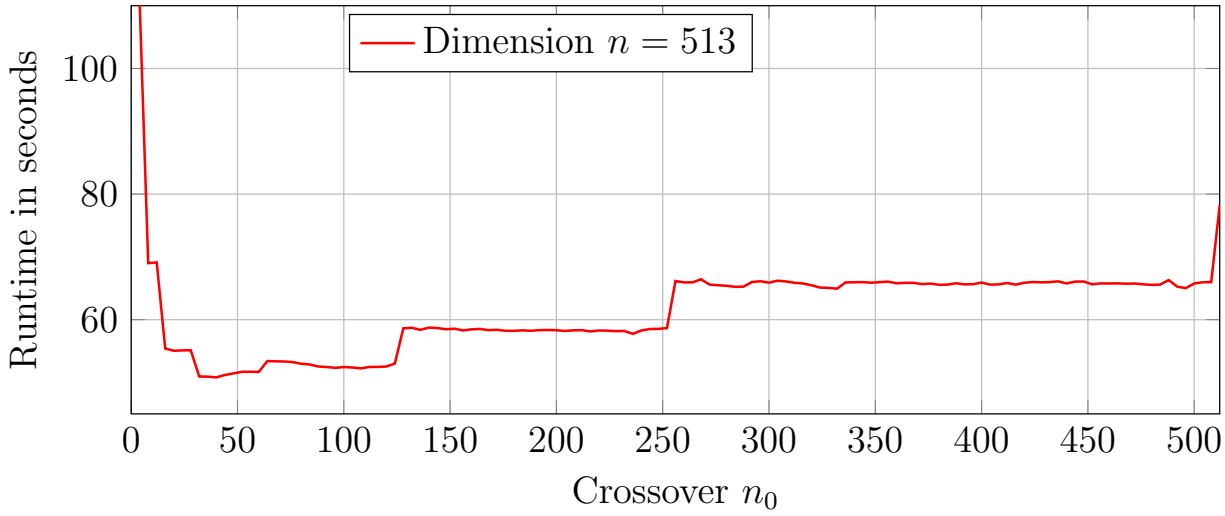
Since we zero-padded the matrices to the next higher power of two (i.e: $n = 512$), our results are similar to what we obtained below for $n = 512$.

3.4 Multiplication of two 512×512 matrices



The runtime of the Strassen's algorithm, without any crossover point, took 296 seconds (averaged over 5 runs) to multiply two 512×512 matrices, while the runtime of the conventional $O(n^3)$ matrix multiplication algorithm took only 9.74 seconds for multiplying the same dimensions. Our results show that the runtime decreased to approximately 7.05 sec between the crossover points of $n_0 = 30$ to $n_0 = 50$, with the runtime saving over the conventional algorithm for $n = 512$ being approximately $(9.74 - 7.05) \times 100/9.74 \approx 27.6\%$. We also note that this crossover range is higher than our analytical crossover point of $n_0 = 16$ for even n.

3.5 Multiplication of two 513×513 matrices

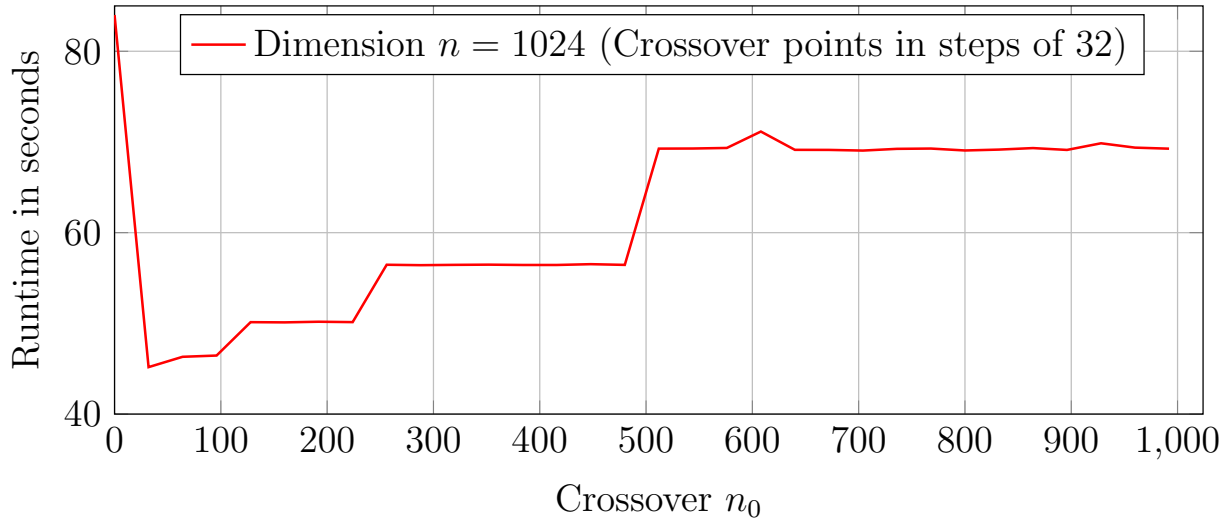


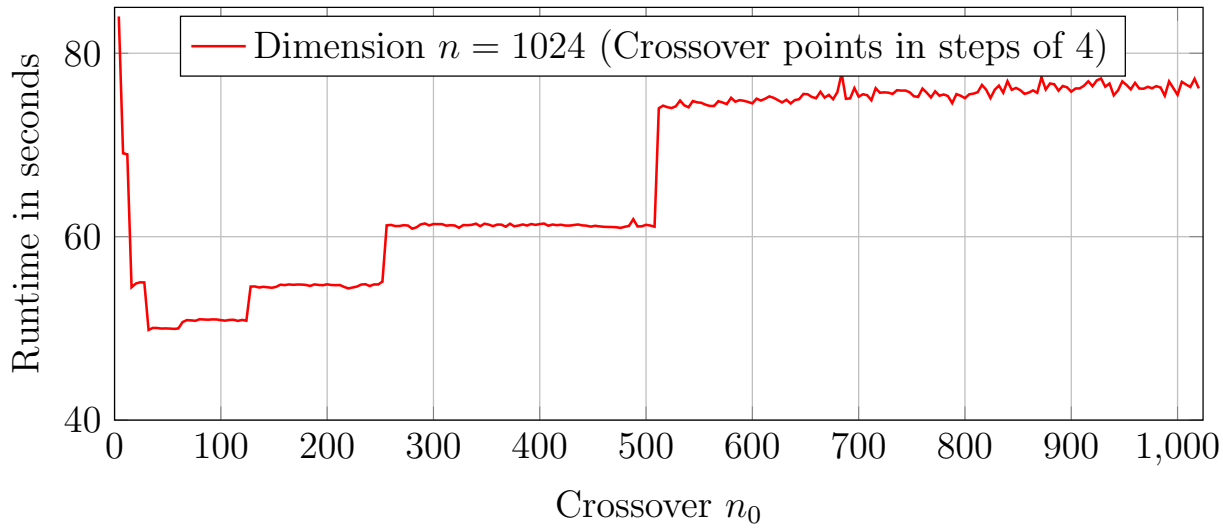
The multiplication of two 513×513 matrices, due to our zero-padding technique, is analogous to multiplying two 1024×1024 matrices.

The runtime of the Strassen's algorithm, without any crossover point, took 2092 seconds to multiply two 513×513 matrices, while the runtime of the conventional $O(n^3)$ matrix multiplication algorithm took only 9.78 seconds for multiplying the same dimensions. Our results show that the runtime decreased to approximately 50.8 sec between the crossover points of $n_0 = 30$ to $n_0 = 50$. We note that, even with the crossover technique, the modified Strasseen's algorithm performs worse than the conventional $O(n^3)$ algorithm. This is because, the conventional algorithm is directly multiplying two 513×513matrices, whereas our modified Strassen's algorithm is multiplying two 1024×1024 matrices.

3.6 Multiplication of two 1024×1024 matrices

Following are the two plots that we obtained from the experiments to multiply two 1024×1024 matrices using Strassen's algorithm on two different machines. The plots shows the results when the crossover point incremented in steps of 32 and steps of 4 respectively.

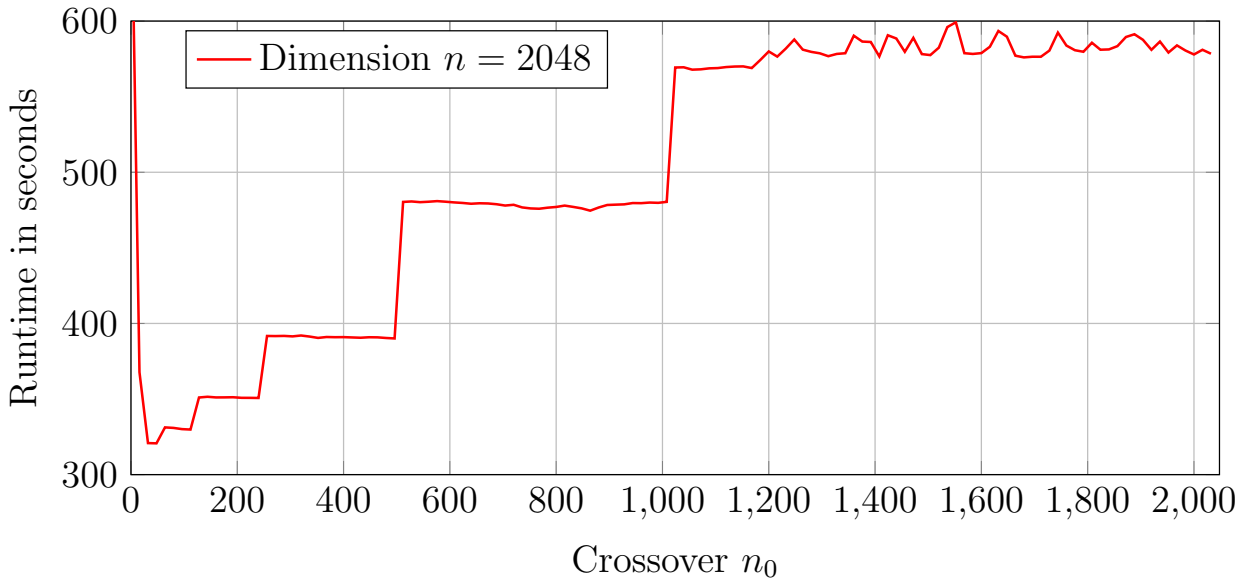




For the input matrix sizes of 1024×1024, Strassen's algorithm completes the matrix multiplication at 49.83 seconds with $n_0 = 32$ while the conventional algorithm takes approximately 84.01 seconds. This is a saving of approximately $(84.01-49.83)\times 100/84.01 \approx 40.44\%$ improvement in runtime. The original, un-modified Strassen's algorithm on our MacBook air using Python took nearly 2022 seconds to multiply two 1024×1024 matrices.

3.7 Multiplication of two 2048×2048 matrices

When the matrix sizes are 2048×2048, we found the optimal crossover point to be $n_0 = 32$, at which the runtime was found to be around 321 seconds, while the conventional algorithm took nearly 834 seconds on our Python implementation.



4 Discussion on the Improved Strassen's Algorithm

We implemented a modified version of Strassen's algorithm, where the algorithm will switch over to the conventional $O(n^3)$ matrix multiplication algorithm at an optimal crossover point $n = n_0$. Our implementation of Strassen's algorithm can be used to multiply both even and odd sized matrices. We considered two different strategies to pad an odd-sized matrix. The first one was to add just one row of 0's and one column of 0's whenever we encountered an odd-sized matrix during the recursion of Strassen's algorithm. The second strategy was to identify the next larger power of two and pad the input matrices with sufficient number of columns of 0's and rows of 0's to make their sizes a power of two. The resultant matrix was obtained by only copying the entries for $i = 1, \ldots, n$ and $j = 1, \ldots, n$, from the padded result matrix. In both cases, our experiments yielded the same optimal crossover points for all matrix sizes that we considered. We found the runtimes of the first strategy were

somewhat higher than the runtimes that we observed using the second strategy. Therefore, we decided to adopt the second strategy for padding odd-sized matrices.

We determined the optimal crossover point by conducting a set of experiments, where we varied the matrix sizes and the crossover points. The results were presented in the previous section, "Experimental Results and Analysis" in detail, and have been summarized in the table below. All the times indicated in the table are in seconds. The last column shows the percentage of improvement seen in the modified Strassen's algorithm over the conventional algorithm, whenever there was an improvement. In the absence of any improvement, the corresponding entry in the "% Improvement" column was left blank in the table below.

Size n	Conventional	Original Strassen's	Improved Strassen's	% Improvement
131	0.16	42.6	0.86	-
256	1.11	12.8	0.87	32.6%
257	1.14	297	7.25	-
512	9.74	296	7.05	27.6%
513	9.78	2092	50.8	-
1024	84.01	2022	49.83	40.44%
2048	835.59	14708	321.00	61.58%

When we tested our modified Strassen's algorithm against smaller matrices of size $n \leq 128$, the $O(n^3)$ conventional algorithm outperformed Strassen's algorithm. This is because for smaller values of n, the $O(n^2)$ term in the recurrence equation for Strassen's algorithm requires $18 \cdot n^2$ operations, which is a reasonably large value. Therefore, the dominance of $18 \cdot n^2$ is more apparent when we compare the performance of Strassen's algorithm against the conventional algorithm for smaller values of n.

However, as we increased the matrix sizes to $n \geq 256$, we started seeing improvement in runtime. The performance improvement was quite significant when the modified Strassen's algorithm was run with the optimal crossover points. In order to find the optimal crossover point experimentally, we ran Strassen's algorithm in a for loop, each time incrementing the crossover threshold. The plots of our experimental results in the "Experimental Results and Analysis" section above show the range of the crossover values for different matrix sizes. The optimal crossover values for different matrix sizes are shown in the table below.

Matrix size n	Experimental crossover value n_0
131	30 - 50
256	30 - 50
257	30 - 55
512	30 - 50
513	30 - 50
1024	32 - 64
2048	32 - 64

Our experimental results for the optimal crossover values are higher than the analytical results that we derived in the "Introduction" section. There could be multiple reasons for

this discrepancy. The most important reason is that we assumed that the cost of any single arithmetic operation (such as addition, subtraction, multiplication, and division) is 1, and other operations such as copying matrix entries from one to another are free. However, this was a very simplistic assumption. Many of these operations are dependent on n and cannot be always considered constant. Moreover, the cache hits and misses may not always be consistent between various operations and therefore it is not always correct to assume that the above operations have constant runtime. For this theoretical assumption to hold true, we needed to conduct our experiments on much larger matrix sizes than what we used. In spite of many optimizations, the largest size that we could conduct the experiments was $n = 2048$, so we could not experimentally identify the individual factors that contributed to the difference between the analytical and experimental crossover values.

5 Triangles in Random Graphs

For any undirected, unweighted graph $G = (V, E)$, we can obtain the number of walks from any two vertices $v_i, v_j \in V$ of length n in the (i, j) entry of matrix A^n. This can be proven by induction, which is ommitted here. However, with this property, the walk for any vertex to itself in a triangle with two other vertices will have a length of 3. Therefore, with A^3, we can obtain the number of triangles in graph G by adding all the diagonal entries and dividing by 6 since any three counts of walks of length 3 will be one triangle, and each walk will be counted twice, as there would be 2 directions for each vertex in each triangle.

For this problem, we used the modified Strassen's algorithm that we implemented earlier. We first created a 1024×1024 matrix A with all zero entries. Using `numpy`'s `random.choice()` method, we initialized each entry $A[i][j]$ to either 1 or 0 based on the input probability p. If the entry $A[i][j]$ was set to 1, then the diagonally opposite entry $A[j][i]$ was also set to 1. This is needed as our adjacency matrix represents an undirected graph, which means if there is an edge between vertices u and v, then the adjacency matrix contains both edges (u, v) and (v, u).

Having created a random 1024×1024 matrix A as discussed above, we can now compute A^3 using our modified Strassen's algorithm. We decided to use $n_0 = 32$ as the crossover point since $n_0 = 32$ was found to be an optimal crossover value for $n = 1024$. Since our matrix represents an undirected graph, it is symmetric across its diagonal. We called our modified Strassen's algorithm twice to compute $C = A \cdot A \cdot A$ and computed the sum of the diagonal entries of the resultant matrix, C. Finally we divided the diagonal sum by 6 and returned the results, which is the number of triangles in our undirected graph. The expected number of triangles in an undirected graph with 1024 vertices, as given in the problem, is $\binom{1024}{3}p^3$. Five sets of experimental results for the given probabilities are shown below.

5.1 Experiment 1

p	Expected no of triangles	Number of triangles from experiment
0.01	178	164
0.02	1427	1440
0.03	4817	4791
0.04	11419	11536
0.05	22304	22413

5.2 Experiment 2

p	Expected no of triangles	Number of triangles from experiment
0.01	178	185
0.02	1427	1445
0.03	4817	4599
0.04	11419	11135
0.05	22304	22386

5.3 Experiment 3

p	Expected no of triangles	Number of triangles from experiment
0.01	178	195
0.02	1427	1437
0.03	4817	5083
0.04	11419	11089
0.05	22304	21466

5.4 Experiment 4

p	Expected no of triangles	Number of triangles from experiment
0.01	178	189
0.02	1427	1352
0.03	4817	4799
0.04	11419	11017
0.05	22304	21842

5.5 Experiment 5

p	Expected no of triangles	Number of triangles from experiment
0.01	178	172
0.02	1427	1399
0.03	4817	4656
0.04	11419	11355
0.05	22304	22715

5.6 Number of Triangles: Average of the Experimental Results

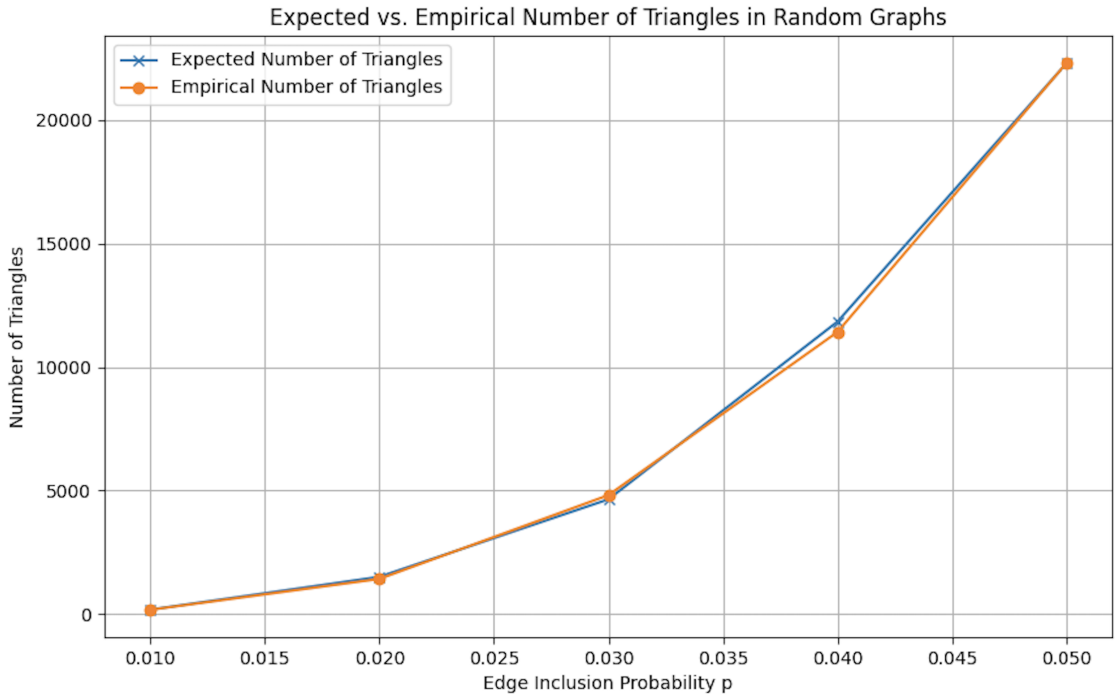
The average number of triangles (rounded to the nearest integer) for the above five experiments is shown below. We have also listed the percentage of the absolute difference between the number of triangles from the experiment and the expected number of triangles computed using $\binom{1024}{3}p^3$.

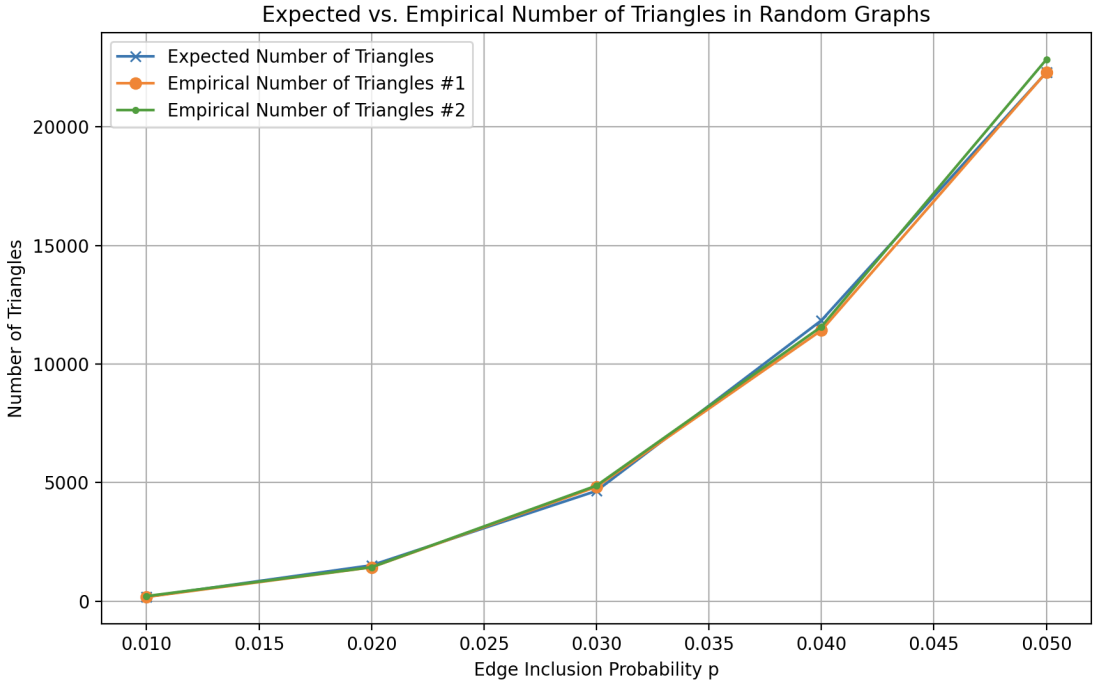
p	Expected no of triangles	Average no of of $\triangle$'s from experiment	|% Difference|
0.01	178	181	1.68%
0.02	1427	1415	0.84%
0.03	4817	4787	0.62%
0.04	11419	11226	1.69%
0.05	22304	22164	0.63%

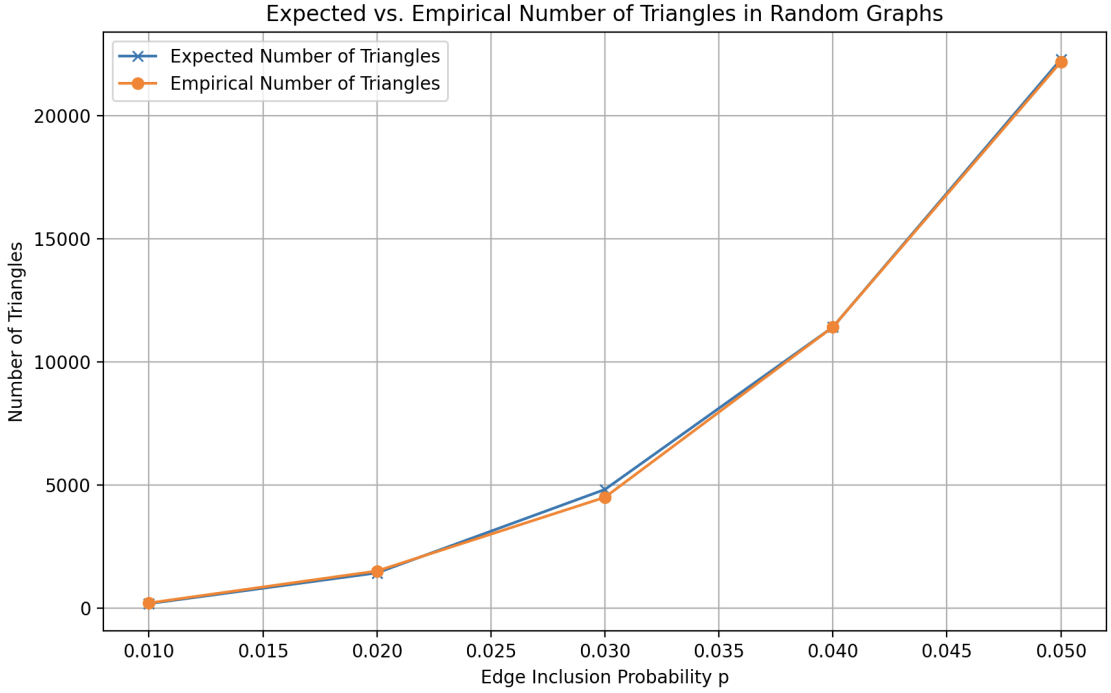
6 Experiments on a different Machine

We repeated the above experiments on a different machine. The table below summarizes both our experimental results and the expected theoretical counts, calculated using the expected number of triangles denoted by $\binom{n}{3}p^3$.

p	Expected	Results 1	Results 2	Results 3	Results $\bar{x}$
0.01	178.433	184	212	203	199
0.02	1427.464	1512	1432	1503	1482
0.03	4817.69	4648	4871	4500	4673
0.04	11419.714	11839	11574	11421	11611
0.05	22304.128	22294	22840	22198	22444




Expected vs. Empirical Number of Triangles in Random Graphs
Expected Number of Triangles
Empirical Number of Triangles #1
Empirical Number of Triangles #2
Number of Triangles
0
5000
10000
15000
20000
0.010
0.015
0.020
0.025
0.030
0.035
0.040
0.045
0.050
Edge Inclusion Probability p


Expected vs. Empirical Number of Triangles in Random Graphs
Expected Number of Triangles
Empirical Number of Triangles
Number of Triangles
0
5000
10000
15000
20000
0.010
0.015
0.020
0.025
0.030
0.035
0.040
0.045
0.050
Edge Inclusion Probability p

While the count for $p = 0.01$ closely matches theoretical predictions, the magnitude of difference between the average of the calculated values and the analytical expected values increase as p increases. The ratio of magnitude of difference to the values themselves is smaller. Moreover, as p increases, the graphs become denser, and the results converge around the theoretical expectations. Even though there is some variability with the triangles in randomly generated sparse graphs, the variability revolves around the theoretical expectation suggesting that with denser randomly generated graphs, the optimized Strassen algorithm can accurately calculate the number of triangles in a graph.