Programming Assignment 1 Report

Vanessa Chambers and Chenai Mangachena*

(Dated: April 2020)

I. THEORETICAL ANALYSIS FOR CROSS OVER POINT

The conventional method of multiplication will be denoted by C. Let n denote the size of the matrix input.
This requires 8 multiplications and 4 additions. Multiplication takes $O(n^3)$ time while addition takes $O(n^2)$ time. This results in a closed form :
$C(n) = 2n^3 + n^2$ (with aid of the internet)

Strassen's method replaces some multiplications in the conventional method with some additions, and since it's generally faster to do additions than multiplications, for very large values of n, Strassen's method is faster. Strassen's method will be denoted by S. Strassen's method involves 7 products as well as 18 additions/subtractions. Recursively, the runtime of Strassen's is then denoted by $S(n) = 7S(\frac{n}{2}) + 18(\frac{n}{2})^2$

The goal is to find the crossover point denoted by n_0.
n_0 is the point for which when $n > n_0$, Strassen's is faster and when $n < n_0$, the conventional method is faster. When $n_0 = n$, this is the point when the runtime of switching to conventional (denoted by $T(n)$ and that of running the conventional method $C(n)$ is equal. We can find this point by substituting into Strassen's recurrence, instead of $S(\frac{n}{2})$, with the closed form of the conventional, giving the runtime at this specific point to be:

$$T(n_0) = 7(2(\frac{n_0}{2})^3 - (\frac{n_0}{2})^2) + 4.5(\frac{n_0}{2})^2$$
$$C(n_0) = 2n_0^3 - n_0^2$$

From this we can deduce that n_0 is approximately 15.

* CS124, Harvard University.

II. STRASSEN'S ALGORITHM CODE IMPLEMENTATION

Implementation:

We used Java to write a program that optimizes matrix multiplication using Strassen's algorithm. We modified our implementation of Strassen's algorithm such that it takes in three inputs: matrix 1, matrix 2, and a crossover point. At this crossover point, the algorithm will switch from Strassen's matrix multiplication to standard row by row, column by column matrix multiplication. In order to find the most optimal crossover point for a specific dimension matrix, we ran Strassen's algorithm with several crossover points, starting from 2, and ending at dimension(matrix). Each crossover point was tested five times per dimension and was timed during each run; we averaged these running-times for five trials to observe how running time changed for different crossover points (Table 1). After data was collected, the crossover point of the modified Strassen's algorithm was plotted against running time, and this curve was compared to the running time of the naive matrix multiplication running time (Figure 1). From this curve, we determined the optimal crossover point for each dimension tested, and these values were recorded and plotted as well (Table 2, Figure 2).

Results:

Using the mentioned testing methods, we were able to test multiple crossover points and find the optimal crossover point among them. For matrix dimension < 16, the naive implementation was faster on average than any use of Strassen's; for matrices with dimension d, such that $16 < d \leq 32$, the optimal crossover point was 16; for matrices with dimension d, such that $32 < d \leq 512$, the optimal crossover point was 32; for matrices with dimension > 1024, 64 was the optimal crossover point.

Analysis:

For low-dimension matrices, Strassen's algorithm did not show to be much more efficient at any breaking point (dimension 1 - 16). However, as dimension increased by several powers of 2 (32 - 512), the optimal breaking point seemed to be located near 32. For 1024x1024 dimension matrix multiplication, the optimal breaking point was 64. These optimal breakpoints were then plotted against matrix dimension, and excel was used to

generate the logarithmic regression

$$\text{crossover point} = 9.2332 \cdot ln(dimension) - 11.733 \tag{1}$$

to fit the data. However, when using this regression to try to generate crossover points, the equation did not produce the most optimal points for matrices of all dimensions. Thus, we concluded that when the matrix dimension was plotted against the optimal breaking point, we could not determine any suitable regression that could fit and approximate the data well. (This is also evidenced by the visibly erratic nature of the graph in figure XX). This is most likely due to the fact that in the interest of time, we only tested matrix sizes and breaking points that were powers of 2; this limited the accuracy of our test. To solve this problem, it would be useful to test crossover values that were not powers of 2; this could possibly allow us to find a regression that helps us fit and predict the optimal break point for any matrix multiplication. Additionally, it would be helpful to test matrices with dimensions larger than 1024x1024 in order to see how the optimal crossover changes over a larger range of data.

Because we used the immutable array data structure in Java, we had to declare a new array any time we wanted to modify the length of the array. This includes when we divide up the array for Strassen's Algorithm, when we combine the separate parts to form the resulting matrix, or when we pad odd dimensional arrays with 0s. It might be useful to instead use an ArrayList data structure, which would allow us to modify length without allocating additional space with each operation.

Our determination of crossover points allowed us to optimize the running-time of Strassen's algorithm. Online sources reported that in addition to our algorithm implementation, crossover points are also affected by the processor of the computer that the program is run on.

Tables and Figures:

	Average running time (nanoseconds)										
Cross point (horiz)/ matrix dimension (vert)	None (naive)	2	4	8	16	32	64	128	256	512	1024
4	7324.2	43126.8	9026.6								
8	21221	234100.6	75260	30166.4							
16	171320.2	1616934.8	301954.2	89840.8	34456.2						
32	253908.2	7034102	1494877	588132.8	350680.75	299496.8					
64	1735814.2	25915330.2	8017693.2	2451089.5	1122909.6	693453	840263.4				
128	4429024.4	156496782	43309809	11279755.8	5738975.2	4736348	4625864.6	5240283.2			
256	40332009.2	979883397	266767766	88961918.4	39090806.3	33087796.4	35535840.4	37394140.8	62156387		
512	395416406	6165581321	1290404315	524275829	278952496	235475880	267651548	262900169	303276697	662621548	
1024	5082744061	4.3503E+10	1.1405E+10	3728227309	2028892321	1732680217	1722221816	1837406753	2149905533	3192437621	6333254564

Table 1: Average Running time for different crossover points

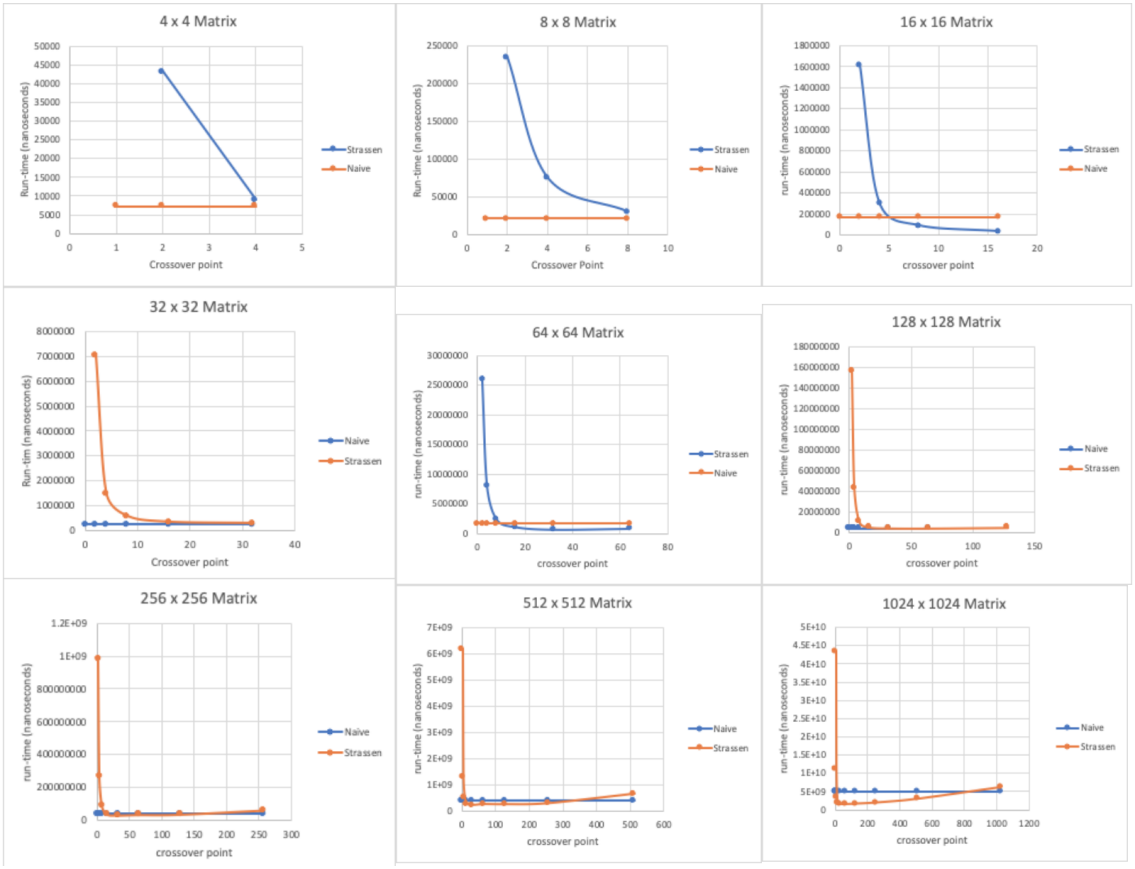

FIG. 1. Average Running time for different crossover points

dimension	4	8	16	32	64	128	256	512	1024
optimal break	0	0	16	32	32	32	32	32	64

Table 2: Dimension vs optimal crossover point

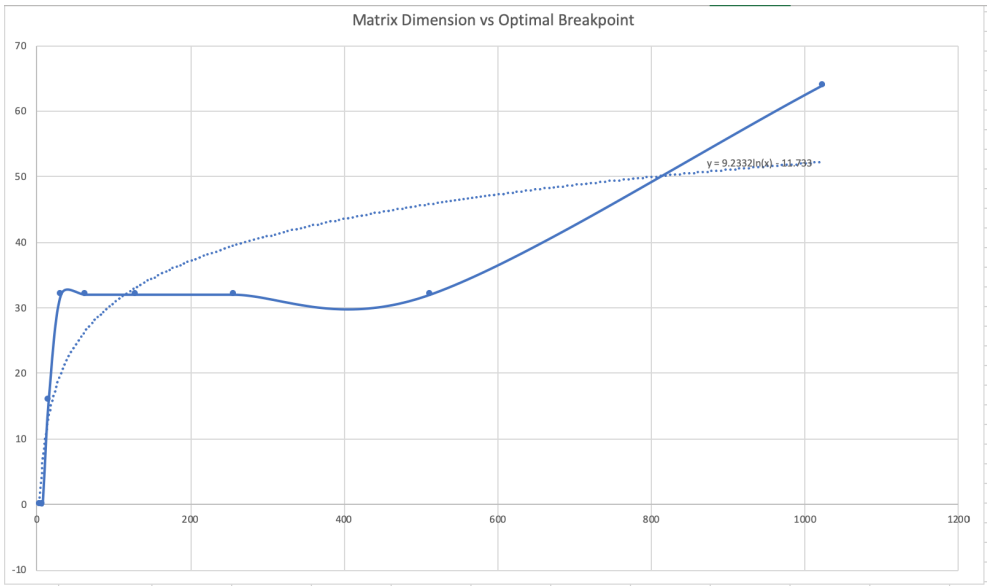

FIG. 2. Dimension vs optimal crossover point

III. TRIANGLE IN RANDOM GRAPHS

For this problem, we chose to represent a graph using an adjacency matrix. Each entry represents an edge between two vertices, and it is populated with a 0 if that edge exists or a 1 otherwise. To determine if there was an edge between two points, a random number was generated between 0 and 1, and if this number was less than the probability of edge inclusion, then we would fill that spot in the matrix with a 1. Otherwise, the spot was filled with a 0. We then used our Java implementation of Strassen's algorithm from Section (2) to find the number of triangles in the random graph. We repeated this process five times for each probability and averaged the number of triangles counted per trial. Evidenced by Table 3 and Figure 3, our experimental results were very close to the expected results (determined by calculating $\binom{1024}{3}p^3$).

There are some drawbacks to our implementation. The slight discrepancies in our values could be attributed to the fact that we only used a pseudo random number generator to generate numbers, which were then used to determine if an edge was included or not. The pseudo-randomness of this number generator could result in an uneven probability that a certain value is produced, thus affecting the likeliness of producing a number $\leq p$. Additionally, it seems like a waste of space to use an entire 1024x1024 array to represent the graph, given that a large number of the entries are redundant–this program could be made better by finding a way to optimize this storage method; for example, maybe only using a matrix half the size to store relevant values could be a better implementation, however this would require modification of Strassen's algorithm (or could be entirely incompatible) to support this matrix.

Probability	Expectation	Results
0.01	178.433024	181.6
0.02	1427.464192	1451.4
0.03	4817.691648	4850.8
0.04	11419.713536	11419.4
0.05	22304.128	22030.8

Table 3: Expected vs Experimental Number of Triangles

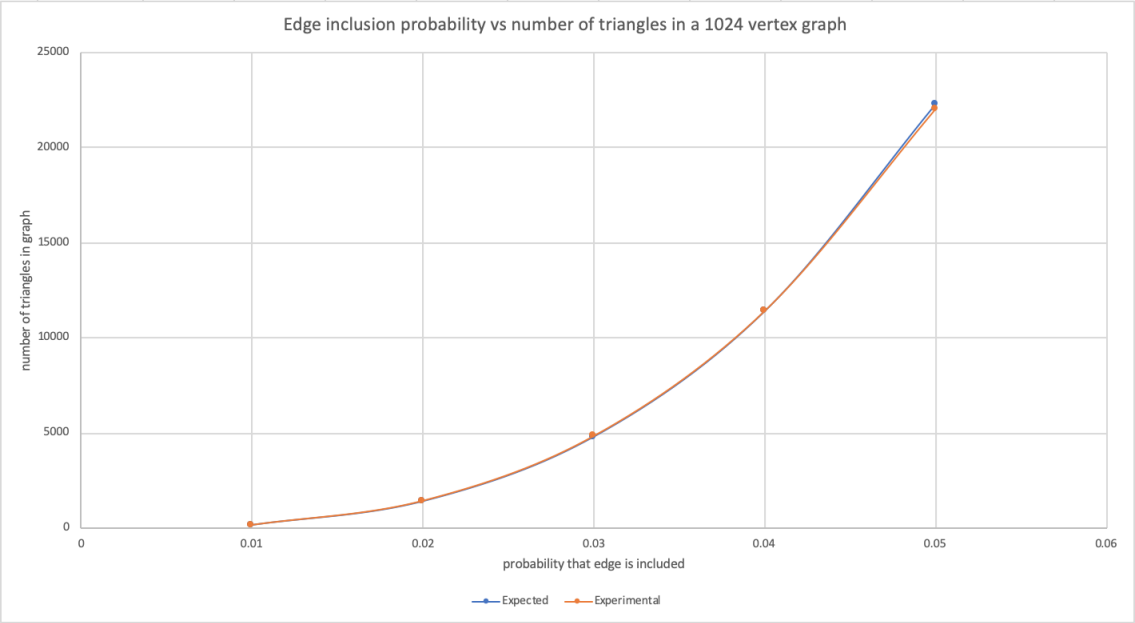

Figure 3: Plot of Expected vs Experimental Number of Triangles