Programming Assignment 2

Methods

Optimizations were primarily utilized for implementing Strassen's algorithm as efficiently as possible, even though development for the standard method was also completed along with solving for the number of triangles on randomized graphs. For reference, we refer to the first matrix as a and the second matrix as b.

For the standard matrix multiplication method, we simply implemented element-by-element matrix multiplication; ie. the element (i, j) of the product matrix n is $\sum_{k=1}^{n} a_{ik} b_{kj}$. For Strassen's algorithm, we compute the product matrices and following the general pseudoalgorithm provided in lecture but with the following program improvements:

1. In order to prevent excessive copying of data (which would slow down performance at high n), we do not make new matrices for the eight submatrices $A - H$. Instead, we write a wrapper class (Matrix) that wraps the int data array and returns object references to the original matrix as submatrices. This wrapper class contains the original integer, the edge coordinates for the different submatrices, and a getValue() method to return the value at a particular relative submatrix location.

2. We do not provide matrix addition and subtraction methods; instead, since there are only a constant number of matrix additions and subtractions per iteration of Strassen's recursive call, we group all them together into one for loop and thus eliminate unnecessary separate iterations.

3. For those matrices such that n is odd, we pad the matrix by 1 additional row and column with zeroes, which should not affect the actual matrix multiplication in the submatrix of interest. This is far more efficient than padding to the nearest power of 2 (which while would guarantee a viable submatrix dimension, is extremely space inefficient), and the relevant matrix product can then be obtained by returning the first n rows and columns.

The base case for Strassen's algorithm is a matrix of size 1, in which we simply return the product of the two numbers in a and b.

In order to compute the number of triangles in randomized graphs, we create an integer array, iterate through the array, generate a value between [0, 1], and fill in with 1 if the value generated is less than or equal to the probability specified. We then compute the cube of the matrix and compute the number of triangles in the manner specified in the problem statement (averaged over 5 trials for each probability).

Discussion

In order to first solve for the cutoff point analytically, we recognize that the runtime of the standard matrix multiplication algorithm should be $2n^3 - n^2$, as for each row of a we iterate through each column of b, and for each column we perform n multiplications and $n-1$ additions. The runtime of Strassen's should be $7T(n/2) + 18(n/2)^2$, as we recursively perform Strassen's 7 times for each call and 18 matrix additions/subtractions, each of which takes $(\frac{n}{2})^2$ time. For odd n, we must pad by 1 row and column; thus, for odd n, the runtime should be at least the runtime of $n+1$. Then, the cutoff should be a point n such that the next iteration using Strassen's provides the same running time as the runtime using standard matrix multiplication. We first solve for the cutoff analytically based on the following equation in Mathematica:

$$2n^3 - n^2 = 7(2(n/2)^3 - (n/2)^2) + 18(n/2)^2$$

Solving this gives $n = 15$. However, for odd integers, we must pad and solve a different equation:

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

which gives $n = 37.17$, assuming padding takes 0 time. This is a rough approximation to reality, as padding in its fastest implementation would consist of creating a matrix of size $n+1$ then reassigning pointers to the original matrix in the corresponding positions. However, these solutions are not exact; the first solution of $n = 15$ assumes that every matrix can be subsequently divided in half, which is only true for even numbers and so the cutoff is rather $n_0 = 16$, and the second solution of $n = 36$ assumes that every matrix can be padded and then subsequently divided in half, which is only true for odd numbers and so with rounding is $n_0 = 39$.

In order to test these answers in practice, we run efficient implementations of Strassen's algorithm and the conventional algorithm. We define our iterative runs the same way as the equations: that is, to look for the value of n_0 such that a next round of Strassen's (following standard matrix multiplication on a matrix of dimension $\frac{n}{2}$ or $\frac{n+1}{2}$, depending on whether or not n is even or odd) results in a lower runtime than using standard matrix multiplication for some n. The results can be seen in Figure 1; for each data point (ie. cutoff dimension), we run the comparison on 3 matrices with randomly generated integers ([0, 1], [0, 2], and [-1, 1]) and average the runtimes for better generalization. Furthermore, for each datapoint, we repeat the simulation 100 times in order to significantly reduce the variance in results as these runtimes are very small for low n.

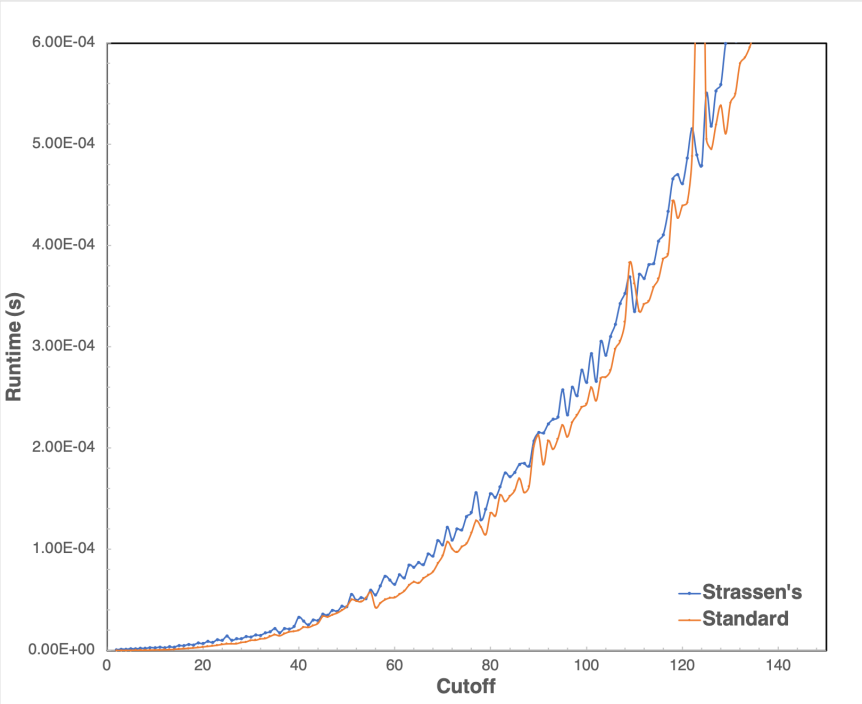

Figure 1: Runtime vs Cutoff n for both standard and modified Strassen's matrix multiplication algorithms with randomized entires in [0, 1]

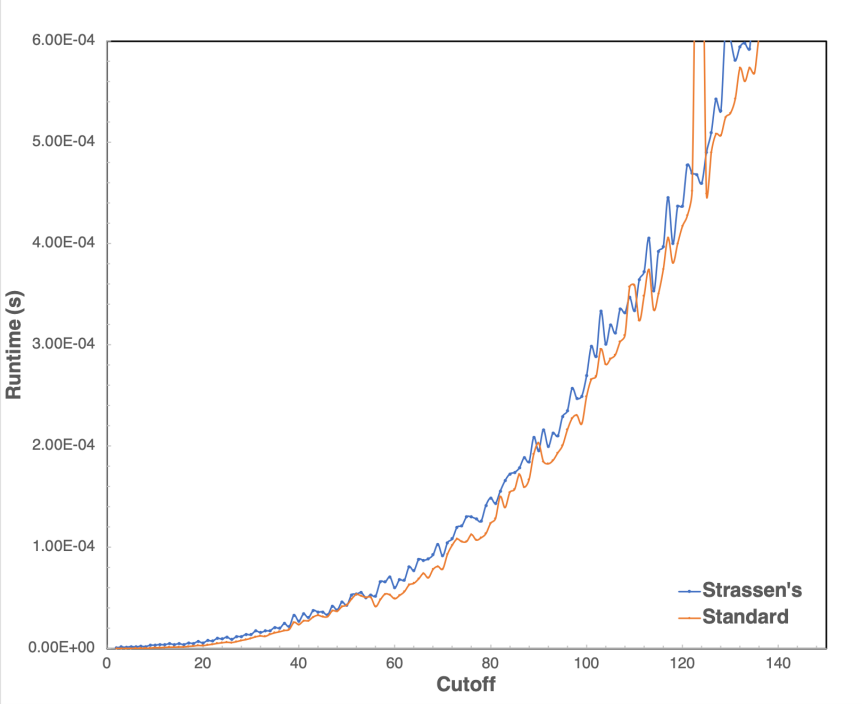

Figure 2: Runtime vs Cutoff n for both standard and modified Strassen's matrix multiplication algorithms with randomized entires in [0, 2]

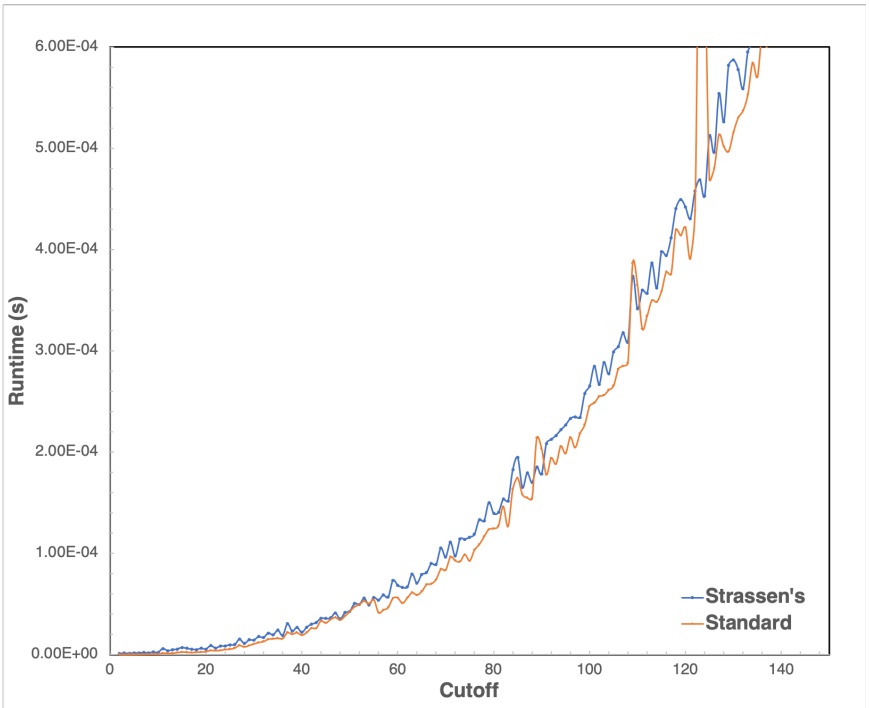

Figure 3: Runtime vs Cutoff n for both standard and modified Strassen's matrix multiplication algorithms with randomized entries in [-1, 1]

Entry value	Odd	Even
[0, 1]	50	109
[0, 2]	50	111
[-1, 1]	54	89 (109)

Table 1: Cutoff values for different ranges of entry values

The cutoff point of interest in both Figures 1 and 2 will be where the line for the modified Strassen's algorithm first intersects the line for the standard matrix multiplication algorithm. At this point, the runtime using a round of Strassen's algorithm becomes equally or more efficient than simply using standard matrix multiplication. We see that the simulated cutoffs for randomized entry values are largely the same between [0, 1] and [0, 2]. This is expected, as any difference arising between these two would be due to differences in integer addition between two very small and positive values, which we do not anticipate. However, for randomized entries in [-1, 1], we see that while the odd cutoff largely remained the same, the even reduced significantly. Since this is unexpected (since we expect Java adds signed integers the same way regardless of sign) especially for small numbers, we look more closely and see that an additional cutoff arises at $n_0 = 109$, in much closer agreement to the previous entry values. In this way, and because we do not see any deviation with

odd value entries, it is likely that the cutoff should not vary based on integer value itself (provided it stay within overflow range).

While these values are different that what is theoretically computed, the difference can be likely traced back to original assumptions in solving the analytical equations. We initially ignored any differences in runtime between integer arithmetic, whereas in reality addition/subtraction and multiplication/division have different runtimes. Furthermore, the algorithm ignores complexities introduced in implementation, including memory I/O. Whereas the standard matrix multiplication algorithm is n^2 in memory (excluding the inputs), Strassen's algorithm utilizes more space than the standard matrix multiplication algorithm because of the many submatrices that need to be computed in addition; this extra runtime overhead is not factored into the analytical expression for optimized Strassen's.

Finally, we test the optimized implementation of Strassen's on solving for the number of unique triangles (ie. a cycle with 3 nodes) on randomized graphs, where each edge (from one node to another) is included with some specified probability. We see in the figure below that the computed average number of triangles is very close to what is expected from combinatorics. The equation of the best fit line is $N = 188731059p^{3.02}$, very close to the actual expected value formula.

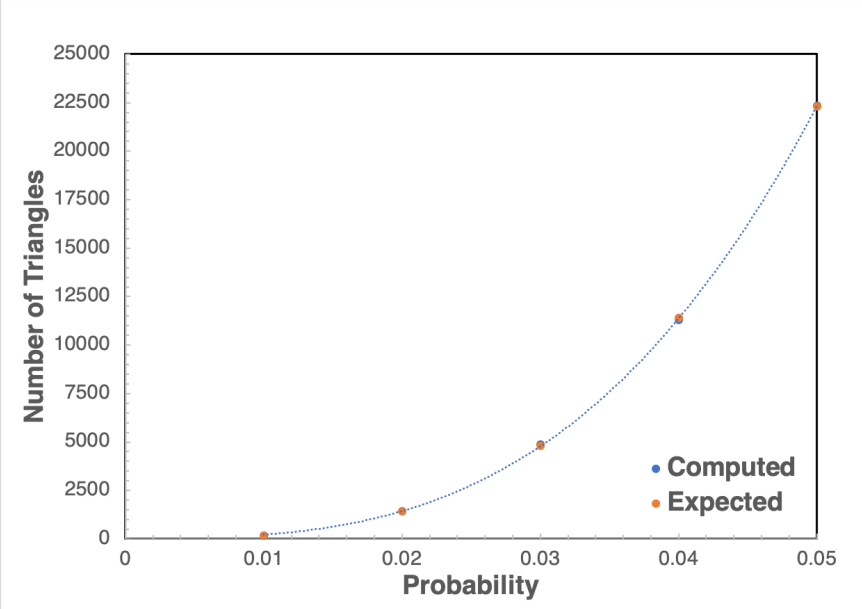

Figure 4: Computed and expected number of triangles vs. p

p	Average N
0.01	179.0
0.02	1434.4
0.03	4869.6
0.04	11280
0.05	22357.8

Table 2: Average computed N for different values of p

References

[1] Java 8 Docs, Oracle.

[2] Computer Science 124, Harvard University.