CS124 Programming Project 2

Sean Ty, David Zhang

February 2021

1 Introduction

We discuss an implementation of Strassen's algorithm to multiply two square matrices, accounting for both time and space complexities. As a recursive algorithm, Strassen's algorithm should have a base case for which the traditional method of matrix multiplication will instead be used. The *crossover point* is then defined to be the point in which Strassen's algorithm switches to the traditional method. To minimize run times, the optimal crossover point is not necessarily equal to 1 or 2 and must be derived–either analytically or experimentally. In this project, we derive it with both ways.

Finally, we use Strassen's algorithm to more quickly simulate the expected number of triangles in a random undirected graph with 1024 vertices, where each edge is generated independently and exists with probability p for $p \in \{0.01, 0.02, 0.03, 0.04, 0.05\}$ and compare this with the well-known result of $\binom{1024}{3}p^3$ expected number of triangles. In this project, Python is the programming language of choice.

2 Analytic Analysis

Let $T(n), S(n)$ be the run times of the traditional and Strassen algorithms respectively for matrices of size n. Since $S(n) \in o(T(n))$, there does indeed exist a point in which $T(n) > S(n)$ for all $n \geq N$ for some $N \in \mathbb{N}$. Thus, after this threshold N, it would always be more efficient to perform Strassen's algorithm. For this experiment, we want to determine whether or not there is a threshold where it is always more efficient to perform the traditional algorithm than recursing with Strassen's. To find this crossover point, it suffices to find the largest n for which $S(n) > T(n)$.

Now note that Strassen's algorithm performs multiplies two $\lceil n/2 \rceil \times \lceil n/2 \rceil$ matrices 7 times and adds/subtracts matrices of the same size 18 times, so if we choose to recurse Strassen, it follows that

$$S(n) = 7S(\lceil n/2 \rceil) + 18\lceil n/2 \rceil^2.$$

But if the traditional method is faster, we would instead perform the traditional method over Strassen's for the 7 submatrix multiplications. Therefore, our recurrence is instead

$$S(n) = 7\min(T(\lceil n/2 \rceil), S(\lceil n/2 \rceil)) + 18\lceil n/2 \rceil^2.$$

Then note that on and before the crossover point we must have $T(n) = S(n)$, so we can substitute $S(n) = T(n) = n^2(2n-1)$ into the above recurrence to get

$$n^2(2n-1) = 7\left\lceil\frac{n}{2}\right\rceil^2\left(2\left\lceil\frac{n}{2}\right\rceil - 1\right) + 18\left\lceil\frac{n}{2}\right\rceil^2.$$

Now we have two cases: if n is even, let $n = 2k$ for some positive integer k. Then we obtain $16k - 4 = 14k + 11$, which yields $k \geq 8$, and so $n \geq 16$. If n is odd, we let $n = 2k+1$ for some nonnegative integer k, the equation yields

$$(2k+1)^2 \cdot (4k+1) = 7(2k+1)\cdot(k+1)^2 + 18(k+1)^2,$$

from which a short trip to WolframAlpha yields $k \geq 19$, and so $n \geq 39$.[1]

Therefore, for matrices with even size the crossover point, as defined by the assignment handout, is 14, while for matrices with odd size the crossover point is 37. Because of this, we would expect roughly $n = 39 - 2 = 37$ to be the theoretically optimal crossover point under our previous assumptions. One caveat of this however is that this does not consider issues such as the time it takes to store submatrices, to extract them, etc., so the optimal crossover value may be even higher.

3 Implementation

3.1 Padding/Unpadding

Strassen's algorithm, which works by recursively halving the matrices, requires a more careful implementation for matrices with odd dimension. To "split" a matrix with odd dimension, we resort to padding. We do this by appending a row and column of zeros at the bottom and the right of the matrix. This makes the matrix have even dimension, after which we can now halve it to perform Strassen's algorithm.

Therefore, Strassen's algorithm now works as follows: given an $n \times n$ matrix we wish to multiply, if n is even, we halve it into four submatrices. Otherwise, we pad the matrix as described above and halve it. We then recursively call Strassen's on the smaller matrices until we reach the crossover point, after which we multiply the matrices using the traditional method.

Another option for padding is to pad until the initial matrix size is a power of 2, although we believe this to be highly inefficient in terms of space since this can require creating matrices much larger than neccessary (since we would have to expand the matrix to the next largest power of 2). Padding in our way allows us to instead only use matrices of size at most $n + 1$, so we chose this method instead.

To prove that Strassen's method still works, we proceed by induction on n. The base case, in which n is at most the crossover point, is true by definition since Strassen's would be the same as the traditional method. For our inductive hypothesis, we assume that Strassen's method works for all $k < n$. Now suppose multiply two $n \times n$ matrices. If n is even, then splitting the matrices is the usual Strassen's algorithm, which we know works due to properties of block matrix multiplication:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}.[2]$$

[1] Or the fact that the left side is larger than the right for sufficiently large values of k since the highest degree term has a higher coefficient and increment k until the left is larger, whatever floats your boat!

[2] Note that each entry A_{ij},B_{ij} here is a matrix.

Otherwise, if n is odd, then we pad a last row and column of 0's in each matrix. Using the same notation as above, this is equivalent to adding a column of zeroes to the right of the submatrices $A_{12}, A_{22}, B_{12}, B_{22}$ and a row of zeroes to the bottom of the submatrices $A_{21}, A_{22}, B_{21}, B_{22}$ (and also one at the bottom-right corner of both factors, the matrices A and B). It suffices to prove that the resulting matrix is the desired product of the unpadded matrices, together with a row and column of zeroes at the right and the bottom, since the algorithm simply removes these zeroes and returns the resulting submatrix.

To prove this, we first note that the last row of the resulting product must be all zero. This is because A_{21}, A_{22} have all zeros in their last row, so taking the dot product of this with everything else still gives zero. The argument is similar for why the last column of the resulting product must all be zero. Finally, the other elements are the same as they would have been had we not padded, since for each element in the product, we are only adding an extra $+0 \cdot 0$ term, where the 0 comes from both the last column of the first matrix and the last row of the second matrix (and everything else is unchanged). This proves that the algorithm does indeed return the product of the two matrices.

Note that this padding method takes time, as we would need to copy the matrices to add a new element. As a result, we would expect the experimental crossover point to be noticeably larger than the analytic crossover point we derived earlier.

3.2 Traditional Cache Optimization

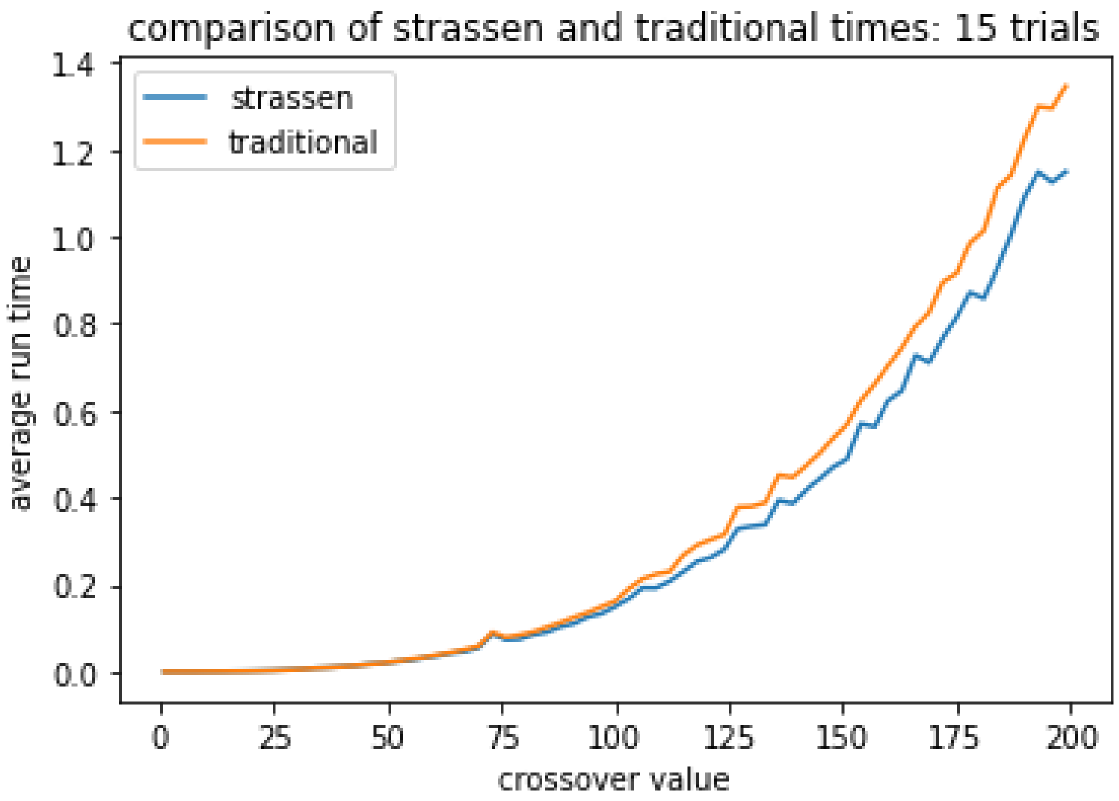
Traditional matrix multiplication is commonly done by taking dot products of rows and columns. In our implementation, to take advantage of cache issues, we instead "partially" take dot products of rows with rows and filling up the matrix by incrementing these dot product values to the resulting matrix. We have done this so that the program accesses contiguous blocks of memory for consecutive operations, which is faster than accessing disjoint locations in memory. This speeds up computation time. One caveat, however, is that Strassen's algorithm will now have a larger crossover value than if we had implemented the dot product method of multiplication instead, since $T(n)$ is effectively shrunk by some factor c. As we will see below, the Strassen crossover point was larger than expected, and we hypothesize that this may be one reason why, together with the issue we have with copying arrays during implementation.

4 Experimental Analysis

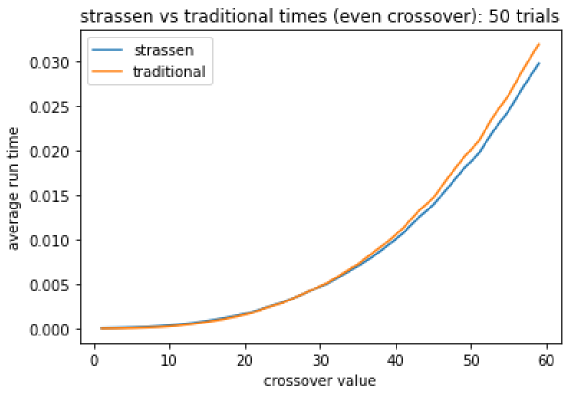
After implementing Strassen's algorithm with our padding mechanism for odd-sized matrices, we then experimentally determined the optimal crossover point for our implementation. In order to do this, we begin with the observation that if n is under the crossover point, it must be more efficient to directly traditional multiply the matrices than to split with Strassen's and then traditional multiply. We expect our optimal crossover point to be no more than 200,[3] so to determine the optimal crossover point, we looked at the run times for multiplying two $n \times n$ matrices for two given algorithms: the traditional algorithm, and Strassen's where we divide the matrix exactly once. We only need to consider dividing it by exactly once because by definition, the crossover point must be the minimum value for which dividing is less efficient.

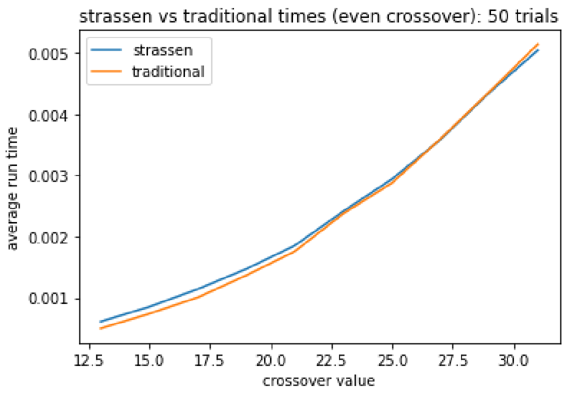
[3]We will also see from the graph later that the traditional method already starts becoming slower for all n, so we are already over the optimal crossover.

For each $2 \leq n \leq 200$, we multiplied two $n \times n$ matrices 15 times using both algorithms and took the average run time of each. This is done to minimize the variance in run time that may have been derived from other sources (e.g. start up issues). The plot for the run times can be found below:



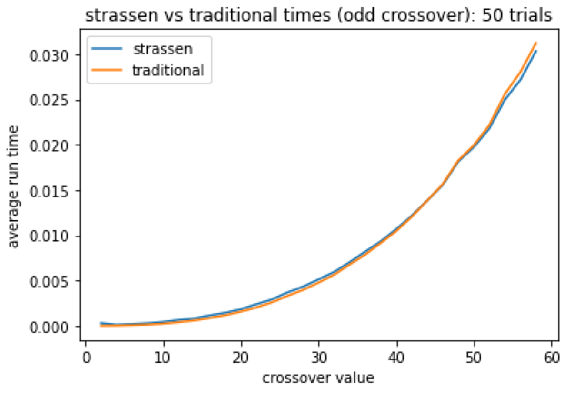
Fortunately, we see that the Strassen's method is indeed asymptotically better than the traditional method, as evidenced by how the blue line is lower than the red line for larger sizes. From the plot above, we see that the crossover point is at most 60. So, to find the optimal crossover value in this range, we performed 50 repetitions and took the average for each method in this range of sizes. Since the analytic crossover points were different for the odd and even cases, we will examine these cases separately. We first examine the even case. The left graph below shows the performances of the two algorithms in the range [0,60], and the right graph is the zoomed up version of it near the optimal crossover point:

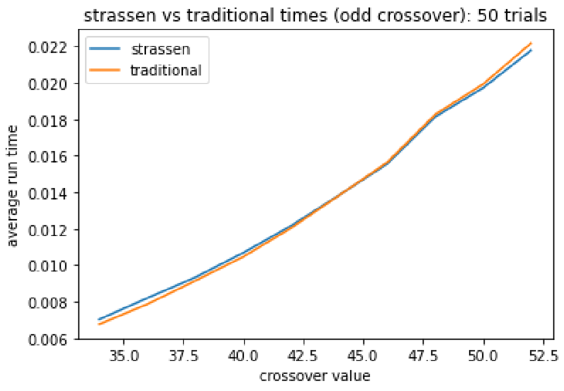




From the plots above, we see that our optimal even crossover point for our implementation is about $n \approx 26$. We then checked this using the arrays used to store the run times, and found that $n = 24$ is the optimal even crossover point.

We now examine the odd case. As before, the left graph below shows the performances of the two algorithms in the range [0, 60], and the right graph is the zoomed up version of it near the optimal crossover point:





From the plots above, we see that our optimal odd crossover point for our implementation is about $n \approx 45$. We then checked this using the arrays used to store the run times, and found that $n = 43$ is the optimal odd crossover point. Combining these results, we see that $n = 43$ is our optimal crossover point.

We recognize that in both cases, our experimental crossover points are larger than our analytic crossover points. This is likely due to implementation details that were not taken into account when we computed analytically. For one, memory allocation takes time, especially given how new matrices of size $\frac{n}{2}$ are created recursively. Another possible explanation is also as we have discussed before: perhaps our efficient implementation of the traditional multiplication algorithm scaled down its run time, making it more difficult for Strassen's algorithm to compete with for smaller matrices. We also think that the experimental values are close to the analytic values because of possible inefficiencies when implementing the traditional multiplication algorithm.

In our experiments, we multiplied integer matrices, since our implementation requires integers. We did this because we thought some issues might arise when we deal with floating point values (such as approximations, and perhaps our constant time operations no longer being constant).

5 Triangles in Bernoulli Random Graphs

5.1 Random Graph Generator

We needed to generate the graph edges in such a way that each edge is generated independently from the rest with probability p for some $0 < p < 1$. Therefore, if we consider the adjacency matrix of the graph, this is equivalent to each element above the main diagonal to be independent and identically distributed as Bern(p). Therefore, it suffices to generate a Bern(p) random variable.

To generate the random variable, we use the `random` method of Python's Random library. This method generates a continuous[4] uniformly distributed random variable in the interval $[0, 1)$. While it does not generate a 1, the probability of a 1 is zero anyway so this does not change anything. Then, since the generated number is uniform, the probability that the number is less than p must be exactly p, so the event $\{X < p\}$ where X is our generated number is distributed as Bern(p).

Finally, for each entry above the main diagonal, we use this Bern(p) random variable to the entry: if $X < p$ the entry equals 1, and if $X \geq p$ the entry equals zero. The other entry completely opposite it from the main diagonal is then assigned the same value (since the graph is undirected).

5.2 Results

Once the graph was generated, we counted the number of triangles by cubing the adjacency matrix, summing the diagonal entries, and dividing the sum by 6. This works because when we cube the adjacency matrix A, the entry in the row i column i denotes the number of paths of length 3 that start and end at vertex i.[5] This method of summing the diagonals, however, overcounts the number of triangles by a factor of 6–one for each vertex (and there are three) and one for each orientation (clockwise or counterclockwise). This implies that the algorithm indeed counts the number of triangles. Finally, we perform this with 50 trials and average the number of triangles in each trial and return the mean. We have done this because given enough trials, this converges almost surely to the true mean.

We used Strassen's algorithm for the matrix multiplication, since it runs faster than the traditional method for our size of $n = 1024$. The results of the simulation are summarized in the table below:

p	Simulated	Expected
0.01	178.62	178.43
0.02	1427.08	1427.46
0.03	4824.92	4817.69
0.04	11420.09	11419.71
0.05	22313.00	22304.13

We notice that the simulated values are very close (less than 1% away) to the expected values, which is good because it suggests the number of trials we performed is large enough. With more trials, the simulated values should approach the expected values almost surely due to the Law of Large Numbers.

[4]Pretty much continuous for our purposes.

[5]We can see this from the definition of matrix multiplication, since this would mean that row i column j of A^n denotes the number of ways to get from i to j in exactly n steps.