CS 124 Programming Assignment 2: Strassen's Algorithm

Leonard Tang, Jackie Wei

Spring 2022

Contents

1 Abstract

We implement a hybrid version of Strassen's algorithm for matrix multiplication. In the process, we analytically derive an optimal threshold (i.e. crossover point) for switching from a pure implementation of Strassen's algorithm to the well-known standard matrix multiplication algorithm. We empirically search for an optimal practical crossover point by implementing our hybrid algorithm in Go and experimenting on various input sizes and thresholds. Finally, we use Strassen's algorithm to model and count triangle paths in a randomly-generated graph of 1024 vertices.

2 Deriving an Analytical Crossover Point

First, recall that for a $n \times n$ matrix, Strassen' Algorithm computes 7 subproblems (sub-multiplications) of $n/2 \times n/2$ matrices and 18 additions/subtractions of $n/2 \times n/2$ matrices. We can thus use the following recurrence to describe the number of operations for a $n \times n$ input:

$$T(n) = 7 \cdot T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 18 \cdot \left(\left\lceil \frac{n^2}{4} \right\rceil\right)$$

Note that we make use of the ceiling function here in the cases where n is odd. Indeed, if n is odd observe that $T(n/2)$ is not well-defined. To obtain a tractable recurrence, we elect to *pad* our matrix with an outer layer of zeroes, thus creating an even $(n+1) \times (n+1)$ matrix.

Now for the standard matrix multiplication algorithm, observe that for a $n \times n$ matrix, we must compute $n^2 \cdot n = n^3$ integer multiplications (n integer multiplications for each of n^2 row-by-column vector multiplications). There are $n^2 \cdot (n-1)$ additions/subtractions ($n-1$ additions/subtractions for n^2 row-by-column multiplications). Thus, we can express the number of operations for the input as:

$$M(n) = n^3 + n^2(n-1) = 2n^3 - n^2$$

To derive an analytical crossover point, first observe that for small n, $T(n) > M(n)$; note otherwise that we wouldn't need to define a crossover point and would simply always use Strassen's and never standard matrix multiplication. Thus, we want to find the smallest n_e such that $M(n_e) \geq T(n_e)$. For now, assume that n is always an even positive integer. Then, the ceiling functions in $T(n)$ disappear and we have the following inequality:

$$\begin{aligned} 2n_e^3 - n_e^2 &\geq 7 \cdot T\left(\frac{n_e}{2}\right) + 18 \cdot \frac{n_e^2}{4} \\ &\geq 7 \cdot \left(2 \cdot \frac{n_e^3}{8} - \frac{n_e^2}{4}\right) + 9 \cdot \frac{n_e^2}{2} \\ &= \frac{7n_e^3}{4} + \frac{11n_e^2}{4} \end{aligned}$$

Note the second inequality follows from the fact that $T(n) > M(n)$ for $n < n_e$. Solving for n_e in this inequality (numerically via Python), we have that $n_e = 15$. Concretely, we have that for even values of $n < n_e = 15$, we should switch from using Strassen's algorithm to using standard matrix multiplication.

Now, if n is odd, that is if $n = 2k+1$ for some $k \in \mathbb{Z}_{\geq 0}$, we instead consider computing $T((n+1)/2)$ since $T(n/2)$ is not well defined as discussed above. We want to find minimal odd n_o satisfying the following

inequality:

$$\begin{aligned} M(n) = 2n_o^3 - n_o^2 \geq T(n) &= 7 \cdot T\left(\frac{n_o+1}{2}\right) + 18 \cdot \frac{(n_o+1)^2}{4} \\ &\geq 7 \cdot \left(2 \cdot \frac{(n_o+1)^3}{8} - \frac{(n_o+1)^2}{4}\right) + 9 \cdot \frac{(n_o+1)^2}{2} \\ &= \frac{7(n_o+1)^3}{4} + \frac{11(n_o+1)^2}{4} \end{aligned}$$

Solving numerically, we have that $n_o = 37$ in the odd case. As such, when n is odd, we should switch to conventional matrix multiplication when $n \leq 37$.

3 Empirical Exploration

3.1 On Choosing Go

We initially attempted to code Strassen's algorithm in Python, which worked well for small (i.e. $n < 256$) sizes of matrices. However, we quickly discovered that no matter which crossover point we elected to use, our Python implementation of Strassen's algorithm would not terminate in a reasonable time for matrices of size $n = 1024$ (on the order of approximately 2 minutes). This would have made experimentation much more time-consuming and far less enjoyable. For our own sanity (and also because Leonard is a huge fan of Go), we chose to re-implement Strassen's algorithm in Go, which of course is significantly (≈ 40 times) faster than Python. With our Go implementation, the same $n = 1024$ example took only roughly 2 seconds.

3.2 Optimizations

We optimize our code by minimizing matrix initializations as much as possible. We also avoid unnecessary copying of matrix values; that is, in all appropriate instances (e.g. when padding) we simply manipulate the matrix in place without creating a new matrix.

3.3 Handling the Even and Odd Cases via Padding

Recall that in the previous section we handle even and odd matrix sizes differently in our implementation. We describe our procedure to handle the n odd case here. Explicitly, we pad the input matrix with both an additional row and additional column of zeros before the usual Strassen logic. Critically, observe that this enforces an even matrix size. Then, right before we return the matrix product, we remove the appended row and column of zeros. This yields the desired product for odd-sized matrix input. Note that we perform this padding procedure for all recursive calls to our Strassen function where the input matrix has odd size.

As an alternative option, one can consider performing a singular padding on the initial $n \times n$ matrix such that the size of the padded matrix matches the N, the next largest power of 2. Note that since $N < 2n$, our asymptotic runtime remains unaffected. However, we conjectured that such an expensive padding would increase runtime in a practical setting, as in some cases the input matrix nearly doubles in size, so although one saves the cost of padding down the line, at every recursive call the initially padded matrix will always be greater than or equal to the recursively padded matrix, and we thought that the trickle-down effect from running Strassen's on larger matrices was not worth saving some marginal time padding later. As such, we chose to use the recursive padding scheme for all experiments. This of course means that the optimal crossover point may change given whether n is a power of 2 or not, as we explore later, which is not an issue that arises when padding to the nearest power of 2.

Notice that both procedures are indeed equivalent. Implicitly, they both manifest the following identity (as commonly seen in iterative linear algebra methods):

$$\begin{bmatrix} A & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} B & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} AB & 0 \\ 0 & 0 \end{bmatrix}$$

3.4 Results

To empirically determine the optimal crossover point, we run our Strassen's implementation on two randomly generated matrices with integer entries of 0 to 10. To ensure consistency, all results were run on Leonard's MacBook.

Recall again that we are interested in potentially divergent behavior of our algorithm under the even and odd cases of n. As such, we carefully choose canonical representations of each case.

That is, to investigate our algorithm's behavior of even input n, we choose $n = 2^k$ for some sufficiently large k to make things interesting. The motivation for doing so is that for all recursive calls of our Strassen function, the input will be even (by construction of n). Thus, in some sense, $n = 2^k$ is a "maximally even" input that is an interesting example to study. Concretely, we choose $n = 512, 1024,$ and 2048. Below we display the running times of various crossover points for these "maximally even" points. Note that all runs are averaged across multiple ($k = 10$) trials to reduce noise:

Crossover Point	$n = 512$	$n = 1024$	$n = 2048$
8	0.402s	2.624s	20.131s
16	0.242s	1.499s	11.351s
32	0.170s	1.184s	8.755s
64	0.170s	1.150s	8.438s
128	0.206s	1.334s	10.384s
256	0.298s	2.145s	15.989s

Observe that the above crossover points indeed form a sufficiently comprehensive candidate set. In particular, since the recursive inputs will only ever be powers of 2, crossover points that are exactly powers of 2 are just as informative as any crossover points between powers of 2. From the data, it appears that $n = 64$ is the optimal crossover point. Note that if we had chosen to pad to the nearest power of 2 at the beginning, $n = 64$ would also be the optimal crossover point for odd n as well, although it is not necessarily the case that padding at the beginning is faster overall.

Analogously, to investigate the behavior of odd input n, we choose $n = 2^k + 1$. Again, the motivation here is to find a "maximally odd" input n such that all recursive calls to our function will take in odd inputs. Indeed this is the case for $n = 2^k + 1$, which we can verify with a brief proof.

Suppose $n = 2^k + 1$. Then under our padding scheme the initial input to our Strassen's function has size $2^k + 2$. All recursive subproblems then have size $2^{k-1} + 1$. This is again odd, and we will pad it to be $2^{k-1} + 2$. Iterating this argument, it's easy to see that under our padding scheme all recursive calls will have odd input for initial input $2^k + 1$. Concretely we choose $n = 513, 1025,$ and 2049. Below we display the running times of various crossover points for these "maximally odd" points:

Crossover Point	$n = 513$	$n = 1025$	$n = 2049$
8	1.503s	9.765s	1m16.116s
16	0.514s	3.555s	28.074s
32	0.272s	1.915s	15.086s
60	0.222s	1.505s	11.161s
64	0.199s	1.483s	11.080s
70	0.170s	1.252s	8.964s
80	0.173s	1.234s	9.306s
90	0.171s	1.235s	9.095s
128	0.168s	1.258s	9.357s
256	0.180s	1.397s	10.553s

From the data, it seems like the optimal crossover point for the maximally odd case is roughly 70. Taking into account both the even and odd cases, we suggest an empirical crossover point of 70, as crossover values less than 70 for the odd case are more penalizing than crossover values larger than 64 for the even case (at least in the [64,70] range).

4 Discussion

4.1 Discrepancy Between Theory and Experiments

Recall in our derivations from section 2 that the optimal crossover point was $n_e = 15$ for n even and $n_0 = 37$ for n odd. However, our experimental crossover points were much larger: roughly $4n_e$ for the even case and $2n_o$ for the odd case. While somewhat surprising at a first glance, this makes sense given our implicit assumptions during our initial derivations. That is, we totally assumed that arithmetic operations (in particular, additions, subtractions, and multiplications) all had a cost of 1. In practice, this may not be true as the size of integers increase.

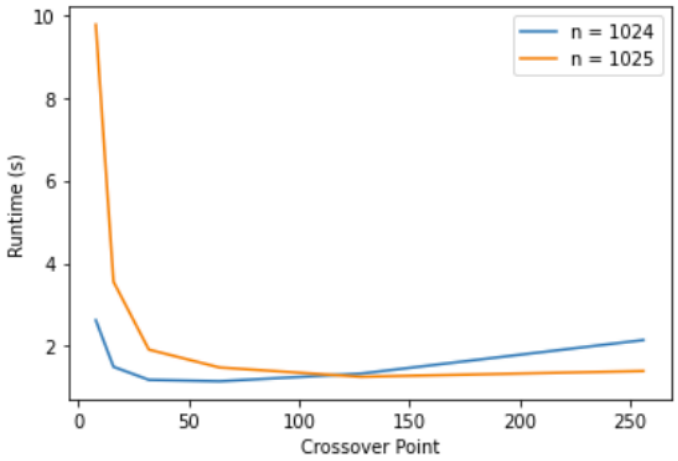
For example, while it may be reasonable that the conventional algorithm runs somewhere near $2n^3 - n^2$ (the only meaningful operation we do not factor in is the creation of the large matrix at the beginning), the Strassen algorithm includes several operations that are not included in the recurrence calculation, such as: allocating and de-allocating memory to create seven submatrices; padding the matrix with zeros; and re-combining the matrices into one large matrix at the end. As a result, by not including these operations, the analytical computation gives the Strassen algorithm a larger comparative benefit over the conventional algorithm, which would result in a lower theoretical crossover point than in practice.

4.2 Even vs. Odd Cases

From our results, it appears that there wasn't a terribly significant difference in optimal crossover point between the even and odd cases. We posit that this is because our implementation of Strassen's algorithm wasn't efficient enough to allow crossover to happen at low enough values of n (e.g. $n < 37$ or $n < 15$). As such, the addition of padding was relatively insignificant compared to the time of all other operations required. With more time and more clever optimization tricks, we hope to investigate this further.

However, do note that for low values of n, our implementation is significantly slower for the "maximally odd" cases compared to the "maximally even" cases; indeed, for roughly $n \leq 32$. This confirms our intuition discussed in Section 2. That is, for smaller values of odd n, it makes sense just to perform standard matrix multiplication, since by using Strassen's you incur an additional cost of padding that would not exist in the "maximally even" case.

Below we plot the difference in runtime for $n = 1024$ and $n = 1025$:



One thing to note is that although the even case runs generally faster than the odd case for low crossover points, for crossover points $n = 128$ and $n = 256$ the odd case runs somewhat faster. This is because– in the $n = 256$ crossover case, for example–the 1024-size matrix reduces to 512, then 256, then uses the conventional algorithm on a 256-size matrix. However, for the same crossover point, the 1025-size matrix will reduce into $\frac{1025+1}{2} = 513$, then $\frac{513+1}{2} = 257$, then finally $\frac{257+1}{2} = 129$ before using the conventional algorithm. Since we showed that the optimal crossover point is lower than 256, this means that using the conventional algorithm at $n = 129$ will be faster than using it at $n = 256$, even when incurring the additional cost of padding. However, at low crossover points, this matters less (for example, the difference in speed of the conventional algorithm for $n = 8$ and $n = 5$ is probably minimal), so in those cases the added padding time will dominate the total runtime.

Finally, our analysis confirms that our hypothesis regarding which padding method would be more optimal was correct. Note that for the $n = 64$ crossover point, an $n = 1025$ matrix multiplication runs in 1.483s. If we padded at the beginning, this matrix would become an $n = 2048$ matrix, which runs in 8.438s (not including the cost of implementing such padding at the beginning). As such, our recursive padding scheme is significantly faster than one single initial padding, even though it results in different optimal crossover points.

4.3 Matrix Element Magnitude

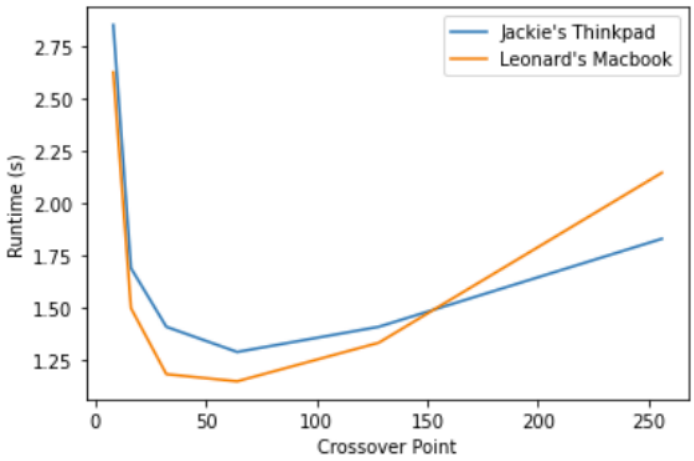
We were curious to see how magnitudes of elements in our input matrices might affect the runtime of our implementation. To that end, we test our implementation on matrices with integers from -10 to 10 and -10^6 to 10^6. As it turns out, matrices of the latter type yielded noticeably slower runtimes. For example, in the $n = 2048$ case, the runtime of our implementation with integers in the range -10^6 to 10^6 compared to the original is as follows:

Crossover Point	$\|M\|_1 < 10$	$\|M\|_1 < 10^6$
8	20.131s	21.228s
16	11.351s	14.595s
32	8.755s	13.465s
64	8.438s	14.373s
128	10.384s	17.534s
256	15.989s	29.119s

This confirms our intuition for runtime scaling properties with respect to integer magnitude as discussed in 4.1.

4.4 Machine Discrepancies

Finally, we hypothesize that machine discrepancies, including cache size and processor types, could affect runtimes (and thus the optimal crossover point). Leonard has a 16GB cache with an 2.4 GHz 8-core processor; Jackie has an 8GB cache with a 2.4 GHz 4-core processor. Below we plot the difference in runtimes for $n = 1024$:



As noted, for most crossover points Leonard's computer performs slightly faster than Jackie's computer, with the only exception being $n = 256$. In this specific scenario, the difference in machines does not change the optimal crossover point, which is still $n = 64$ on Jackie's computer. However, for machines with radically different features, this analysis supports the idea that it is possible that the crossover point may vary. In other analyses we ran, Jackie's computer ran slightly faster for an $n = 128$ crossover point, although the margin was minimal.

5 Finding Triangles

Equipped with an implementation of Strassen's algorithm, we now model the number of triangular paths in a randomly-generated undirected graph. We use $n = 64$ as our crossover point, as tested in prior sections. We write a function `triangle`, which takes as input the probability p that any two points (i, j) have an edge between them. We use a 1024 by 1024 matrix T. For every distinct pair (i, j), with $i \neq j$ (we assume no self edges), we draw a float uniformly at random from 0 to 1; if the value is less than p, we assign $T_{i,j} = T_{j,i} = 1$; otherwise we assign 0. Below we show the average number of triangles (i.e. paths of length 3) averaged across 10 runs compared to the expected number of triangles:

p	Actual Number of Triangles	Expected Number of Triangles
0.01	186	178.4
0.02	1,419	1,427.5
0.03	4,782	4,817.7
0.04	11,495	11,419.7
0.05	21,989	22,304.1

As expected, the actual number of triangles is very close to the expected number, with an error margin of roughly $0 - 5\%$ due to randomness from graph initialization.