cs124 pas 2

Zev Minsky-Primus

March 2021

1 Introduction

Matrix multiplication is usually done with an $O(n^3)$ algorithm: the (i, j)th element of AB is the dot product of the ith row of A and the jth column of B. In this assignment we tested the most common alternative algorithm for matrix multiplication: Strassen's algorithm.

For square matrices of size that is a power of 2, Strassen splits each matrix into 4 submatrices, each with half the number of rows and columns. To multiply two matrices $\begin{pmatrix} A & B \\ C & D \end{pmatrix}, \begin{pmatrix} E & F \\ G & H \end{pmatrix}$, Strassen computes the products $P_1 = A(F - H), P_2 = (A + B)H, P_3 = (C + D)E, P_4 = D(G - E), P_5 = (A + D)(E + H), P_6 = (B - D)(G + H), P_7 = (A - C)(E + F)$. Then, the product of the full matrices is $\begin{pmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_5 + P_1 - P_3 - P_7 \end{pmatrix}$. Since only 7 products of submatrices are needed, Strassen's algorithm is $O(n^{\log 7})$.

However, Strassen's algorithm requires some manipulations which make it slower than regular matrix multiplication for small sizes. We found both experimental and theoretical crossover points where above which it becomes more efficient to use Strassen's algorithm.

2 Base Implementation

2.1 Programming Language

We ended up using OCaml to implement our algorithm. This is important for a couple of reasons. For one, OCaml is a functional programming language, and thus discourages us from effects. While you can still use effects, we chose not to. Secondly, OCaml's variants allowed us to try some optimizations more naturally; more on that later.

2.2 Matrix Representation

Originally, we simply represented the matrix as a struct with a two dimensional integer array, a height field, and a width variable. Obviously the height

and width are easily calculable in $O(1)$ time, but for ease of programming we included them in the struct anyways.

2.3 Multiplication Algorithms

2.3.1 Regular Multiplication

We implemented regular multiplication simply with a couple of loops, calculating cell by cell.

2.3.2 Strassen Multiplication

For Strassen multiplication, we did not do anything fancy at first. We simply divided the given matrices up, called ourselves recursively, and combined the results the way Strassen's algorithm says to.
The only tricky thing we did was to make it work with any sized matrix. In order to do this, at each call to the function, we checked if the width or height for either matrix is odd. We would then copy each matrix into a new matrix which would be the exact same as the old matrix except if either dimension was odd, it would increase that dimension by 1 and make all the new entry's 0.
All multiplication is still compatible because if the width of the first one is equal to the height of the second one before, then because we are doing the same deterministic operation to each dimension, they must still be the same afterwards. And as long as the two matrices are compatible and of even dimensions, all of the multiplications and additions in Strassen's algorithm work.

3 Optimization

When it came to optimizing Strassen's algorithm, our immediate instinct was to attempt to decrease the number of times we copy a matrix. Specifically, there were 3 ways we tried to do this. Ultimately though, all of our optimizations were able to shave off substantially less than 15% of the time, despite having been able to remove (in the asymptote) 29/30 matrix copys.

3.1 Submatricies

We pretty quickly realized that there was no need to copy the whole array when creating a submatrix, as none of the data was changing. Furthermore, because we were doing a pure implementation, we would not have to worry about setting the value in the array for the starting matrix changing the value in the submatrix.
So, we started by changing the representation of a matrix, to also have a point representing the top left point of the matrix. That way, we if we have a 4 by 4 matrix with array a, and we wanted the bottom right 3 by 3 submatrix, we could simply use the same array, and just change the top left to point to the position $(1, 3)$, and change both the height and with to 3.
Then we simply changed the function to get a value from a matrix to take into account where the top left was, an we were done.

Using this method, we were able to remove 9 of the 30 matrix copys in each run of the stars multiplication function. Experimentally, this shaved off around 8% of the runtime.

3.2 Padding

Next, we noticed that we didn't actually need to create a new copy when we wanted to pad the matrices. Once again, we edited the matrix representation. This time, adding two integer fields: one for height padding, and one for width padding. Then our pad function would still create a new matrix with a changed height and width, but rather than changing the array pointer, it would simply change the pad height and pad width fields to the appropriate number. Finally, our get function would now simply check if the row and column specified were in the padded space and if so would return 0. Otherwise, it would do the usual thing.
Using this method, we were able to remove 2 of the 30 matrix copys. Experimentally, this shaved off around 3% of the run time.

3.3 Arithmetic

Finally, for the most complex one. At the start every time we added or subtracted two matricies, we made a copy. From a perspective of trying to minimize

the number of array creation, this was a nightmare, and could be massively improved. How you ask? Well, consider the following pseudo-code that is actually needed for Strassen's algorithm:

bottomRightOutput = subtract (add p5 p1) (add p3 p7)

Here we have to allocate 3 different arrays, once for each add or subtract. But, you could easily make just one new array, and then for each element in that final array with coordinates (row, col), you calculate all at once. Of course, that would require not actually executing the adds before the subtract, and instead have some kind of marker that if someone wants to use it, they need to be added. That way, we could simply allocate new arrays when we absolutely had to.
In practice, we did this by making a new type, AlgMatrix, represented by a variant (for a quick explanation of OCaml variants if you haven't worked with them before, you can go here https://dev.realworldocaml.org/variants.html).
Our variant type was defined as so (with slightly more concise names, but irrelevant):

```
1: type AlgMatrix =
2:   | NormalMatrix of Matrix
3:   | MatrixSum of Matrix * Matrix ;; Star denotes a pair
4:   | MatrixDifference of Matrix * Matrix
```

After creating the definition for algebraic matrices, we re-created many of the functions we use for regular matrices, to suit algebraic ones. For instance we re-implemented get to find the value at the given position from all matrices encoded in the algebraic matrix, and sum them up accordingly. We also implemented a higher-order map function taking an AlgMatrix and another function f, which would create a new AlgMatrix with f applied to each matrix held in the AlgMatrix. Ultimately, this and some other functions allowed us to get rid of 18 more Matrix copys!

At this point I was pretty proud of myself. I had gotten rid of 29 of the 30 matrix copys in the asymptote. So not to toot my own horn, but toot toot.
Then it was time for my victory lap. I ran "dune build" and then "dune exec ./strassen.exe," waiting to see the glorious speed up. Surely we would be at least 30% faster. Maybe even twice as fast! Who knows, maybe we would even be three times faster!
Then, the results printed out, and my heart sank. We hadn't sped up by 30%; in fact, the algebraic matrix had slowed it down! Surely this was wrong, I thought. But after running it again, and again, and again, I had to confront reality: the gambit I once thought brilliant, had failed.

Ultimately, we did not completely abandon this idea completely though. In

order to optimize examples such as the one we gave at the start, we did write a function to allow us to do multiple additions and subtractions with only allocating one new matrix. This might have sped it up on the order of 1% but any difference was so small that it was hard to tell for sure.

So what's going on? Well for one, creating copys of algebraic matrices, even if we are keeping the same array pointers takes time. But more importantly, as we continued to call ourselves recursively, the sizes of these algebraic matrices would build up exponentially with the number of recursive calls! As they would build up, it would take longer and longer to copy the algebraic matrix and to apply functions (such as submatrix) to each regular matrix stored in the algebraic matrix. Furthermore, it would also get longer and longer to get, as the algebraic matrix would get deeper linearly with the number of recursive calls. We started this optimization in order to get the in practice implementation closer to the theoretical one that only takes into account arithmetic operations on numbers. So it's a little ironic, and deserves noting, that just like the problem with that analysis it only focuses on one thing (arithmetic), in trying to remedy it we made a similar mistake by only focusing on memory allocation of arrays.

4 Results

4.1 Mathematical Analysis

The analysis is different for even and odd sizes. Let $f(n)$ be the number of operations needed to multiply two n by n matrices. Calculating this, we have n^2 dot products, each of which require n multiplications and $n-1$ additions, for a total of $f(n) = n^2(2n-1)$.

For an even $2n$ near the cutoff, Strassen's algorithm does 7 multiplications of size n matrices. For two matrices $\begin{pmatrix} A & B \\ C & D \end{pmatrix}, \begin{pmatrix} E & F \\ G & H \end{pmatrix}$, Strassen's algorithm calculates $F-H$, $A+B$, $C+D$, $G-E$, $A+D$, $E+H$, $B-D$, $G+H$, $A-C$, and $E-F$. In addition, it computes $P_5+P_4-P_2+P_6$¡ P_1+P_2, P_3+P_4, and $P_5+P_1-P_3-P_7$, where the P_i are intermediate matrices of size n. This is a total of 18 additions or subtractions of size n square matrices, for a total of $18n^2$ operations. Thus, if $f'(n)$ is the number of operations needed to multiply two n by n matrices using Strassen's algorithm and the recursive multiplications normally, then $f'(2n) = 7f(n) + 18n^2$.

Thus, we have:

$$f(2n) \geq f'(2n) \Leftrightarrow$$

$$4n^2(4n-1) \geq 7f(n) + 18n^2 \Leftrightarrow$$

$$16n^3 - 4n^2 \geq 14n^3 - 7n^2 + 18n^2 \Leftrightarrow$$

$$2n \geq 15$$

For $2n-1$ odd, we need to pad the matrices for Strassen's algorithm, making the matrix have size $2n$. Thus, we have $f'(2n-1) = f(2n) = 7f(n) + 18n^2$. To figure out the cutoff, we do:

$$f(2n-1) \geq f'(2n-1) = f'(2n) \Leftrightarrow$$

$$(2n-1)^2(2(2n-1)-1) = 16n^3 - 28n^2 + 16n - 3 \geq 7f(n) + 18n^2 = 14n^3 + 11n^2 \Leftrightarrow$$

$$2n^3 - 39n^2 + 16n - 3 \geq 0 \Leftrightarrow$$

$$2n - 1 \gtrsim 37.17$$

If we assume that multiplication and addition by 0 is significantly cheaper than generic multiplication and addition, we can further improve.Since the final row and column of D and H are zeros, $F-H, C+D, A+D, E+H, B-D$ and $G+H$ each take only $(n-1)^2$ additions, which corresponds to $6(2n-1)$ fewer additions. Since B, C, F, G have a zero row or column, $A+B, G-E, A-C$, and $E+F$ each take only $n(n-1)$ additions, for a total of $4n$ fewer additions.

Since the last row of $F-H$ and the last column of $C+D$ are zeros, the multiplications $P_1 = A(F-H)$ and $P_3 = (C+D)E$ each take n fewer dot products (since any dot product with the last column will just be 0), for a total saving of $2(2n^2-n)$ operations. Since the last row and column of H and D are zeros, the multiplications $P_2 = (A+B)H$ and $P_4 = D(G-E)$ each require n

fewer dot products, each of which takes 1 less multiplication and 1 less addition, for a total saving of $2n^2(2n-1) - 2(n^2-n)(2n-3) = 2(4n^2-3n)$ operations. Since the last row of $B-D$ and the last column of $G+H$ are zeros, the each dot product of multiplication $P_6 = (B-D)(G+H)$ requires 1 less multiplication and 1 less addition, for a total of $2n^2$ fewer operations.

We also have that the last column of P_1 and P_2 and the last row of P_3 and P_4 are zero. Thus in the final combination, we can save $6n$ additions.

The total savings here are $6(2n-1)+4n+2(2n^2-n)+2(4n^2-3n)+2n^2+6n = 14n^2+14n-6$. Thus, $f'(2n-1) = 7f(n) + 18n^2 - 14n^2 - 14n + 6$. Thus, we have:

$$f(2n-1) \geq f'(2n-1) \Leftrightarrow$$

$$(2n-1)^2(2(2n-1)-1) = 16n^3-28n^2+16n-3 \geq 7f(n)+4n^2-14n+6 = 14n^3-3n^2-14n+6 \Leftrightarrow$$

$$2n^3 - 25n^2 + 30n - 6 \geq 0 \Leftrightarrow$$

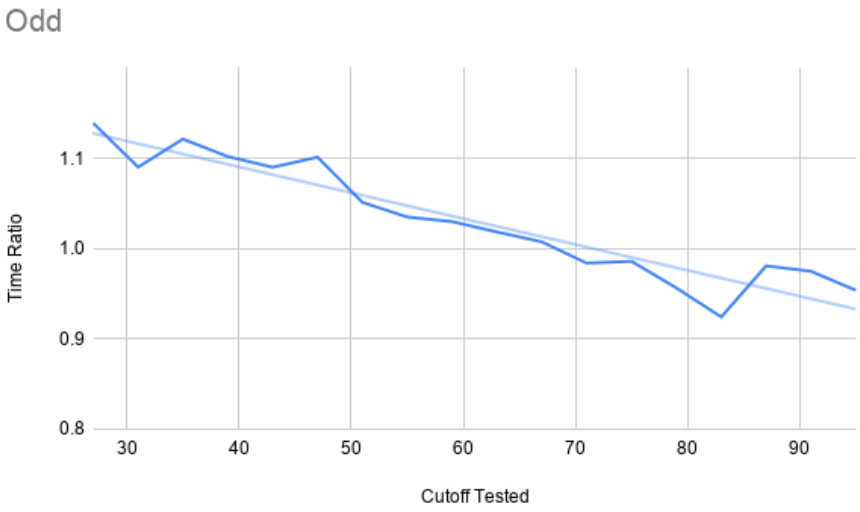
$$2n - 1 \gtrsim 21.4$$

4.2 Experimental Results

In order to test each cutoff c, the goal was to determine the time to completion ratio if we increase c by a factor of 2. We say a factor of 2 because for any given matrix multiplication, any cutoff between dim and $dim/2 + 1$ is the same. For instance, to test the cutoff of 30, we might create two matrices of size $30 * 8 = 240$, and take the ratio of times between doing a multiplication with a cutoff of 30 and a cutoff of 60. A ratio above 1 would indicate that for any matrix of dimension 60, it is more efficient to simply do normal matrix multiplication, and a ratio below 1 would indicate that we should do at least one recursive call to the Strassen multiplication function. Ultimately, this means that the cutoff should be wherever the graph (or the trend line of the graph) hits 1.
We also found that the times to completion differed tremendously from run to run. In order to remedy this as best as possible, we ran 150 tests at each cutoff. For each cutoff, we did one third of the tests with matrices of dimension between 80 and 159, one third between 160 and 319, and one third between 320 and 639.

Below is a graph with the results for even numbers:

And below is the result for odd:



As you can see, there is still quite a lot of deviance. But the trend does seem to indicate that the even cutoff should be 50 and the odd cutoff should be 73. Both these cutoffs are larger than the experimentally determined cutoffs, which makes a lot of sense because the Strassen multiplication does a lot of other operations not accounted for in the theoretical analysis. Most notably, there are many more memory allocation operations.

4.3 Triangle Results

For the number of triangle in a graph problem, here were our results: First of

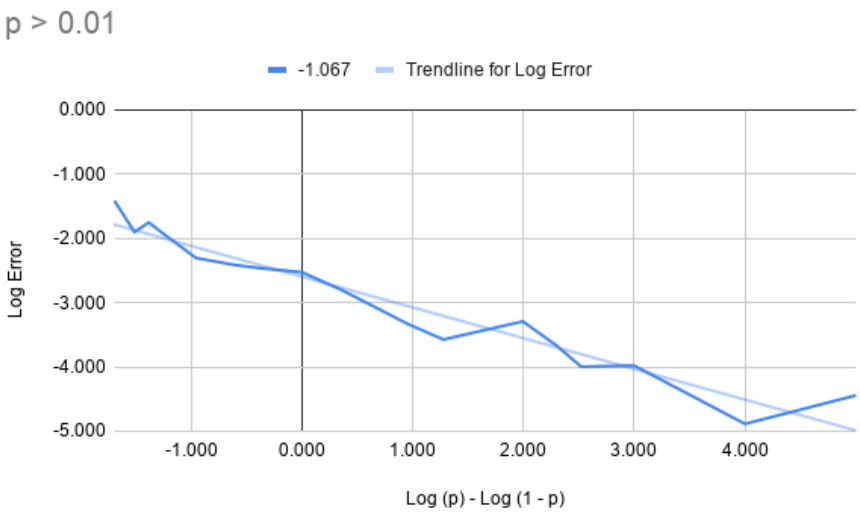
p	Experimental	Expected	% Error
0.01	188	178.4	5.36%
0.02	1437	1427.5	.67%
0.03	4790	4817.7	-0.57%
0.04	11382	11419.7	-0.33%
0.05	22195	22304.1	-0.49%

all, this shows that our algorithm generally comes very close to the expected average; for most of the runs being off by less than 1 percent.
Also, we see is that as the p value get's bigger, the % error reduces. What's most intriguing about this reduction is that it is very very fast at first, and then

slows down very quickly. So, we decided to do 4 runs at each p value, and also do way more different values, to get a better sense of what's going on. Here's the data that what we found:

p	Average Magitude of % Error
0.002	72.595 %
0.004	26.270 %
0.008	8.287 %
0.01	8.565 %
0.02	3.853 %
0.03	1.257 %
0.04	1.765 %
0.05	1.293 %
0.1	0.495 %
0.2	0.385 %
0.3	0.343 %
0.5	0.298 %
0.7	0.155 %
0.9	0.047 %
0.95	0.027 %
0.99	0.051 %
0.995	0.022 %
0.997	0.010 %
0.999	0.011 %
0.9999	0.001 %
0.99999	0.004 %

Something interesting to note about this: at p values that aren't incredibly small the log of the average percent error starts to look linear with $\log(p) - \log(1-p)$. Whether this is just random, or actually fundamental to the nature of the error, I am not sure. An argument that it must be random is that as p goes to 0, the average percentage error can never get larger than 100. This is a result of the fact that the lowest number of triangles we can get is 0. On the next page is a graph that shows this apparent correlation for non-small p values:


p > 0.01
-1.067
Trendline for Log Error
0.000
-1.000
-2.000
-3.000
-4.000
-5.000
Log Error
-1.000
0.000
1.000
2.000
3.000
4.000
Log (p) - Log (1 - p)