Strassen's Algorithm

CS 124 - Sabrina Hu & Smriti Somasundaram

3/29/23

No. of late days used on previous psets: 6

No. of late days used after including this pset: 6

1 Analytical n_0 Value

First, let us find the theoretical crossover point n_0 where we want to stop using Strassen's and switch over to conventional matrix multiplication.

To do so, we want to find the n_0 at which Strassen's takes the same time as conventional matrix multiplication. In other words, we want to find for what n_0 we have

$$T_1(n_0) = T_2(n_0),$$

where $T_1(n)$ is the time recurrence for conventional matrix multiplication, and $T_2(n)$ is the time recurrence for Strassen's.

First, let's find $T_1(n)$ for standard matrix multiplication.

When we do standard matrix multiplication on two $n \times n$ matrices, we get a $n \times n$ matrix as a result, which has n^2 entries we must calculate. To calculate each entry, we perform n multiplications, and then add up each of the n products with $n-1$ additions. Thus, to calculate each entry, we perform $n + (n-1) = 2n - 1$ operations, each of which has time cost 1.

Since we have n^2 entries to calculate, we have:

$$T_1(n) = n^2(2n - 1).$$

Next, let's find $T_2(n)$ for Strassen's algorithm.

When we perform Strassen's on two $n \times n$ matrices, we make 7 multiplications of matrices of size $\frac{n}{2}$, each multiplication of which takes $T_2(\frac{n}{2})$ time. Additionally, when we combine $P_1, ..., P_7$ in Strassen's, we perform 18 additions or subtractions on matrices of size $\frac{n}{2} \times \frac{n}{2}$, each of which takes $(\frac{n}{2})^2$ time.

Thus, we have:

$$T_2(n) = 7T_2(\frac{n}{2}) + 18(\frac{n}{2})^2.$$

To find when Strassen's starts to have a lower cost than standard matrix multiplication, we want to find when $T_1(n_0) = T_2(n_0)$. Since at this theoretical point, we would perform standard matrix multiplication on matrices under size n_0, we have:

$$n_0^2(2n_0 - 1) = 7[(\frac{n_0}{2})^2(2(\frac{n_0}{2}) - 1)] + 18(\frac{n_0}{2})^2.$$

Solving for n_0, we have:

$$\mathbf{n_0 = 15}$$

as our crossing point.

However, this assumes that n is even. When n is odd, we must modify our matrix to be able to perform Strassen's. To do so, we must pad with one layer of 0's to get a $(n+1) \times (n+1)$ matrix (to make the size of our matrix even). When we do this, our equation to find n_0 is now modified to the following:

$$n_0^2(2n_0 - 1) = 7[(\frac{n_0+1}{2})^2(2(\frac{n_0+1}{2}) - 1)] + 18(\frac{n_0+1}{2})^2.$$

Solving for n_0, we get:

$$\mathbf{n_0 \approx 37}$$

as our crossing point.

Thus, we see that our crossing point is $15 \leq n_0 \leq 37$. Above $n_0 = 37$, it should be more optimal to use Strassen's and below $n_0 = 15$, it should be more optimal to use standard matrix multiplication, but between $n_0 = 15$ and $n_0 = 37$, whether Strassen's or standard multiplication will be more optimal varies, depending on the parity and size of n.

2 Experimental n_0 Value

2.1 Optimizations

2.1.1 Padding Zeroes

To deal with odd n and n that are not powers of 2, my initial idea was to pad the $n \times n$ matrix with zeroes up to the next highest power of 2. However, we realized that this was not optimal, especially for n that are barely greater than a power of 2. For example, for $n = 1025$, we would have to add 1023 columns and rows of zeroes to get a 2048×2048 matrix. So, instead, we only padded our matrix with one layer of zeroes when n was odd. Thus, for any n, Strassen's will continue recursing as long as n is even. When the algorithm reaches an odd n, we will add a layer of zeros so that we will have an even size again, and continue. Since we pad only on the outside right and bottom border of the matrix, our multiplication result is unchanged, and we don't use as much padding as the previous method, such that our algorithm can deal with all values of n now.

2.1.2 More Optimized Calculation of $P_1, P_2, P_3, P_4, P_5, P_6, P_7$

The biggest optimization we made was to only use the same two matrices to find the each $P_1, P_2, ..., P_7$ for Strassen's algorith. Instead of finding A, B, C, D, E, F, G, H by splitting our input matrices and storing them into 8 separate matrices, we simply reevaluated the same two $\frac{n}{2} \times \frac{n}{2}$ matrices to find each P_i based on what quarters were needed to find each P_i. For example, since $P_1 = A(F - H)$, we solve for and store A and $F - H$ into just two matrices by finding $F - H$ as we stored it, and we reuse these two matrices to store $(A + B)$ and H for P_2, $C + D$ and E for P_3, etc.

Thus, rather than storing 8 matrices to split our original input matrices, we only use 2. Additionally, we solve for each of $F - H$, $A + B$, etc. in place, thus saving the operation of having to copy both matrices, and then calculating their sum or difference, which would take two more embedded for loops for each pair added/subtracted together. Thus, by using this optimization, we save both space and time complexity.

2.1.3 Using References Instead of Pointers

For both our naive matrix multiplicaation and Strassen's function, rather than passing the two input matrices by value, we pass them by reference. This is more efficient, because since our input matrices can be so big, if we pass them by reference, we do not have to allocate extra memory to copy the arguments but rather simply reference the address of the arguments.

2.2 Results

Table of Runtimes in Microseconds for Standard and Strassen's Matrix Multiplication with Various Crossover Points for n-dimenson Matrices

n	**Standard**	$n_0 = 15$	$n_0 = 16$	$n_0 = 17$	$n_0 = 18$	$n_0 = 21$	$n_0 = 24$	$n_0 = 25$	$n_0 = 32$
31	50.5	230	81.6	87.2	77	76.7	65.2	63.6	**32**
32	34	178	60.4	59.4	59.8	60	51.7	49	**28**
33	39.8	224.4	216.7	71.4	74.1	72.2	61.6	61.2	62.3
63	267	1204	392	387	335	304	**257**	257	292
64	297	1200	394	384	332	**296**	258	256	286
65	286	1500	1194	377	351	319	**277**	277	238
127	1935	4977	**1325**	1363	1269	1171	1168	1163	1254
128	1984	4974	**1344**	1227	1188	1170	1146	1156	1093
129	1982	6076	4068	**1412**	1418	1413	1434	1425	1393
255	14685	21594	**8451**	8376	8185	8171	8195	8241	5029
256	15388	21458	**8176**	8148	8149	8175	8148	8468	5019
257	11236	27044	27252	**10142**	10163	10001	10082	9969	10032
511	95241	155979	**58345**	58426	58394	58103	60757	59931	35226
512	95454	150309	**57325**	57473	57192	57275	57613	57305	35329
513	96680	187771	187682	**69420**	69464	70501	69687	70097	70158
1023	1005730	1062320	**416048**	412374	409328	410014	407717	410834	252227
1024	1015320	1068430	**407087**	406601	412178	415602	417200	413643	251562
1025	805680	1362580	1360870	**506109**	506436	503399	505609	508930	504144

In this table, each time given is the elapsed time for the operation averaged over 10 trials. n is the dimension of the matrix, and n_0 is the crossover point in the modified Strassen's algorithm ran. All times to the right of the second vertical line are times with modified Strassen's algorithm. The algorithms were ran on matrices of size $n \times n$, and each entry of the input matrices were randomly generated to be $0, 1$, or 2. The bolded times represent the first n_0 at which modified Strassen's with that crossover point yielded a faster time than standard matrix multiplication for a given n.

Additionally, note that we intentionally chose to include n in our table that are of the form $2^k - 1$, 2^k, and $2^k + 1$; each one of these forms entails a different amount of padding of zeroes and can show certain discrepancies that come about from this padding, while still covering a wide range of n values.

2.3 Discussion

2.3.1 What is Our Experimental n_0?

In the table above, we bolded the times at which Strassen's with a specific n_0 as a crossover point becomes faster than standard matrix multiplication of matrices of a given size n. The column under which each time is bolded represents an approximate experimental crossover point for the given n.

First, we observe that n_0 is different for various values of n. For example, for $n = 31$, $n_0 \approx 32$, while for $n = 64$, we have $n_0 \approx 21$, and for $n = 255$, we have $n \approx 16$. There is no singular n_0 value that precisely defines the crossover point for all n.

However, there are two ways we can approximate a "general" crossover point. First of all, we can identify an approximate upper bound n_0, such that for virtually all n, Strassen's algorithm with a crossover point at that n_0 runs faster than standard matrix multiplication.

In this case, we see that $\mathbf{n_0 = 32}$ is an approximate experimental upper bound. It is the crossover point for $n = 31$ and $n = 32$, and works as a crossover point for all other n (such that Strassen's is more efficient than standard matrix multiplication).

However, we can also look at another possibility for n_0. As n goes past $n = 127$, we see that a tighter bound on the crossover bound is consistently $n_0 = 17$ as n goes towards infinity. Thus, we can postulate that $\mathbf{n_0 = 17}$ might be a more accurate crossover point for larger n. However, it seems that for virtually all n, our experimental n_0 satisfies

$$\mathbf{16 \leq n_0 \leq 32}.$$

2.3.2 Possible Differences Between Theoretical and Crossover Points

Surprisingly, our experimental n_0 values matched closely to the theoretical ones. While our experimental algorithm might have been accurate for a number of reasons, perhaps because of the optimizations we made, there are several ways that actual implementation of the algorithm could have entailed experimental n_0 values different from the theoretical one.

One of the biggest differences between the theoretical and experimental calculation is that the theoretical one assumes a time complexity of $O(1)$ for addition, subtraction, multiplication, and division operations, and considers all other operations to be free. However, in practice, these operations are not necessarily $O(1)$ time. For example, if we had entries in our matrices that are multi-digit, the actual calculations of multiplying/adding/subtracting the numebrs themselves might take longer (which I talk about in the next subsection).

Additionally, in our program, we have to copy matrices; for example when finding $P_1, P_2, ..., P_7$, we must take the different quarters of M_1 and M_2 to and copy some of them to find these values. While our theoretical calculations dismisses these copying operations as free, in reality, they take a considerable amount of time. However, because we optimized this portion of the algorithm of the program, by minimizing the copying of the submatrices of the input and instead directly copying the results of subtracting or adding the matrices into one matrix, we minimize this extra time that may cause the theoretical and experimental results to differ by a factor of 3; this may be why our experimental results are so close to the theoretical ones.

Finally, discrepancies between theoretical and experimental results can also arise from variations within computers like CPU efficiency and compiler efficiency.

These are just a few possible ways theoretical calculation and experimental implementation could cause differing results. However, our experimental results were very close to our theoretical calculations, which may demonstrate our somewhat minimization of these discrepancies through optimizations.

2.3.3 Different Types of Multiplicand Matrices

We tried looking at the runtime of the algorithms of all three types of proposed matrices: ones with entries randomly chosen out of 0, 1 out of 0, 1, and 2, and out of -1, 0, 1. We did not see any discrepancies in runtime between the three types.

However, -1, 0, 1, and 2 are relatively small numbers. If we were to use entries with larger values they should

eventually yield a slower runtime, since out current calculations assume $O(1)$ time for addition, subtraction, and multiplication, but as entries reach more than one digit, these calculations are not linear time.

2.3.4 Other Observations

Some other interesting observations include the fact that for values of n that are of the form $2^k - 1$ and 2^k, there is a significant drop in time from previous n_0 to the crossover point n_0, while the values of n that are of the form $2^k + 1$ have an almost "delayed" drop in time, in that their n_0 values are often one value above the n_0 values for the $n = 2^k$ and $2^k + 1$ n_0 values. First of all, it is important to note that $n = 2^k - 1$ and 2^k behave very similarly in terms of time complexity, since our algorithm will give a matrix of dimension $2^k - 1$ one layer of padding so that it is size 2^k, which will then allow it to continue Strassen's from thereon out without any new padding.

However, for values of n that are not right under a power of 2, such as those of the form $2^k + 1$, there will be many more rounds of padding, which will cause it to be much less efficient. This especially applies for n of the form $n = 2^k + 1$, where the matrix will have to undergo padding every time Strassen's recurses. Additionally, we see that the reason the times from $n_0 = 15$ to $n_0 = 16$ seem to jump so significantly for all of the n that are of the form 2^k and $2^k - 1$ is because if the crossover point is 16, the algorithm performs standard matrix multiplication when the recursion of Strassen's reaches a matrix of size 16 (which is will for powers of 2). However, if the crossover point is 15, the recursion of Strassen's will reach a matrix of size 16, and then recurse again, which will take more time than just doing standard matrix multiplication. When we calculated our theoretical crossover point, we found that it would take the same amount of time to do standard matrix multiplication at that point as continue Strassen's one more time; however, because of the possible discrepancies discussed, one more round of Strassen's for a matrix of size 16 is a lot slower than standard matrix multiplication at that point.

Thus, this is why there is that significant drop in runtime at $n_0 = 16$ for most of the n that are either a power of 2 or 1 minus a power or 2, and that there are some differences for the n that are 1 more than a power of 2 (because of multiple occurrences of padding).

3 Counting Triangles

3.1 Results

Number of Triangles in a Graph with 1024 Vertices
Calculated with Modified Strassen's with $n_0 = 17$

	p = 0.01	**p = 0.02**	**p = 0.03**	**p = 0.04**	**p = 0.05**
Avg Calculated Number	179.7	1424.5	4733.1	11457	22409.9
Theoretical Number, $\binom{1024}{3}p^3$	178	1427	4818	11420	22304

To calculate the number of triangles in a graph, we used Modified Strassen's with $n_0 = 17$ to find A^3, and then found trace$(A^3)/6$. p gives the probability of including each edge in our graph. We averaged our results over 10 trials for each p. We compared our calculated numbers with the theoretical number of triangles for each given p, given by $\binom{1024}{3}p^3$.

3.2 Discussion

We see that our calculated number of triangles were pretty close to the theoretical numbers, which shows that our Strassen's accurately multiplies matrices.

The small discrepancies in the results can most likely be accounted for by the p aspect. When calculating the theoretical number, we use the expected value of p^3 for all three edges in a triangle being in the graph. However, in our implementation, we generate an each edge with probability p, but our actual ratio of existent edges over all possible edges drawn is not necessarily actually p (which the theoretical calculation assumes is the case). Thus, this is the case for all the varying values of p, so this leads to some small differences between experimental and theoretical result.

4 Conclusions/Reflection

Overall, this assignment was very interesting to see how Strassen's can be used practically with the inclusion of this crossover point. It was interesting to see which optimizations made the biggest differences, and actually get experimental results that were similar to our theoretical calculations. Ultimately, I learned a lot from this programming assignment about Strassen's, divide-and-combine, optimizing the calculations, and differences between theoretical and experimental calculations.