Data Structures and Algorithms

Spring 2014

Cesar Agustin Garcia Vazquez

April 5, 2014

1 Programming Assignment 02

1.1 Introduction

Strassen's divide and conquer matrix multiplication algorithm for n by n matrices is asymptotically faster than the conventional $O(n^3)$ algorithm. This means that for sufficiently large values of n, Strassen's algorithm will run faster than the conventional algorithm. For small values of n, however, the conventional algorithm is faster. Indeed, the textbook Algorithms in C (1990 edition) suggests that n would have to be in the thousands before offering an improvement to standard multiplication, and "Thus the algorithm is a theoretical, not practical, contribution". Here we test this analysis.

Since Strassen's algorithm is a recursive algorithm, at some point in the recursion, once the matrices are small enough, we may want to switch from recursively calling Strassen's algorithm and just do a "base case" of a 1 by 1 matrix, but to switch earlier and use conventional matrix multiplication. Let us define the *cross-over point* between the two algorithms to be the value of n for which we want to stop using Strassen's algorithm and switch to conventional matrix multiplication. The goal of this assignment is to implement the conventional algorithm and Strassen's algorithm and to determine their cross-over point, both analytically and experimentally. One important factor our simple analysis will not take into account is memory management, which may significantly affect the speed of your implementation.

Tasks:

1. Assume that the cost of any single arithmetic operation (adding, subtracting, multiplying, or dividing two real numbers) is 1, and that all other operations are free. Consider the following variant of Strassen's algorithm: to multiply two n by n matrices, start using Strassen's algorithm, but stop the recursion at some size n_0, and use the conventional algorithm below that point. You have to find a suitable value for n_0 - the cross-over point. Analytically determine the value of n_0 that optimizes the

running time of this algorithm in this mode. (That is, solve the appropriate equations, somehow, numerically.) This gives a crude estimate for the cross-over point between Strassen's algorithm and the standard matrix multiplication algorithm.

2. Implement your variant of Strassen's algorithm and the standard matrix multiplication algorithm to find the cross-over point experimentally. Experimentally optimize for n_0 and compare the experimental results with your estimate from above. Make both implementations as efficient as possible.
   The actual cross-over point, which you would like to make as small as possible, will depend on how efficiently you implement Strassen's algorithm. Your implementation should work for any matrices, not just those whose dimensions are a power of 2.
   To test your algorithm , you might try matrices where each entry is randomly selected to be 0 or 1; similarly, you might try matrices where each entry is randomly selected to be 0, 1, or 2, or instead o, 1, or -1. You might also try matrices where each entry is a randomly selected real number in the range $[0, 1]$. You need not try all of these, but do test your algorithm adequately.

Code setup:

So that we may test your code ourselves as necessary, please make sure your code accepts the following command line form:

$./strassen 0 dimension inputfile

The flag 0 is meant to provide you some flexibility; you may use other values for your own testing, debugging, or extensions. The dimension, which we refer to henceforth as d, is the dimension of the matrix you are multiplying, so that 32 means you are multiplying two 32 by 32 matrices together. The parameter input file is an ASCII file with $2d^2$ integer numbers, one per line, representing two matrices A and B; you are to find the product $AB = C$. The first integer number is matrix entry $a_{0,0}$, followed by $a_{0,1}, a_{0,2}, \ldots, a_{0,d-1}$; next comes $a_{1,0}, a1, 1$, and so on, for the first d^2 numbers. The next d^2 numbers are similar for matrix B.

Your program should put on standard output (printf, cout, System.out, etc) a list of the values of the *diagonal entries* $c_{0,0}, c_{1,1}, \ldots, c_{d-1,d-1}$, one per line, including a trailing new line. The output may be checked by a script - add no clutter. (You should not output the whole matrix, although of course all entries should be computed.)

The inputs we present will be small integer, but you should make sure your matrix multiplication can deal with results that are up to 32 bits.

If you expect there will be any problems with running your work on the Harvard nice system (or a standard UNIX system), please contact the TFs early on in the process; also, be sure to include a makefile, or give explicit instructions on how to compile if necessary. Ideally, your program will be compiled, run, and tested by a script, so we would

prefer the following: you would be able to sit in the directory you submit, on nice, type "make", then "./strassen 0 ¡dimension¿ ¡inputfile¿" for some dimension and input file, and have it work.

What to hand in:

Hand in a project report (on paper) describing your analytical and experimental work (for example, carefully describe optimizations you made in your implementations). Be sure to discuss the results you obtain, and try to give explanations for what you observe. How low was your crossover point? What difficulties arose? what matrices did you multiply, and does this choice matter?

Your grade will be based primarily on the correctness of your program, the crossover point you find, your interpretation of the data, and your discussion of the experiment.

Hints:

It is hard to make the conventional algorithm inefficient; however, you may get better caching performance by looping through the variables in the right order (really, try it!). For Strassen's algorithm:

- Avoid excessive memory allocation and deallocation. This requires some thinking.
- Avoid copying large blocks of data unnecessarily. This requires some thinking.
- Your implementation of Strassen's algorithm should work even when n is odd! This requires some additional work, and thinking. (One option is to pad with 0's; how can this be done most effectively?) However, you may want to first get it to work when n is a power of 2 - this will get you most of the credit - and then refined it to work for some more general values of n.

2 First steps

For this assignment, the programming language used was Java SE 1.7 due to the familiarity with the language. All the development and running of experiments are done on a MacBook Aluminum late 2008, with a 2 GHz Intel Core 2 Duo processor and 8 GB 1067 MHz DDR3 in RAM.

The first thing to do is program the straightforward algorithm to multiply 2 matrices. This is going to work as benchmark of Strassen's algorithm. In this section, I only consider square matrices whose size is $n = 2^k$. The idea of the programming assignment is to find a way to make Strassen multiplication look faster than the standard multiplication, considering an n not too big. For this reason, I am only considering the size of the matrices up to $n = 1,024$.
The scenarios that I am considering are: matrices with only 0s and 1s, matrices with

only 0s, 1s and 2s, matrices with only 0s, 1s and -1s and matrices with values in the real interval $[0, 1]$.
The matrices are randomly generated and the running time show in the tables is in milliseconds and it is the average of running the test cases 3 times. I chose 3 times because in the first experiments that I ran, I wanted to compute matrices with size up to 32,768, but that was taking too long, so instead of using the average of 5 as in the previous assignment, I decided to reduce it to 3.

The Table 1 shows the results of running these experiments, from which we can see that up to matrices of size 32, which gives and idea of when to switch from Strassen's algorithm to the standard method. The straightforward algorithm performs exactly n^3 multiplications and after allocation the space for the result, no more space memory is used.

Table 1: Straightforward multiplication algorithm run with different test cases and sizes

Test\n	2	4	8	16	32	64	128	256	512	1024
{0, 1}	0	0	0	0	0	1	7	76	1,175	15,102
{0, 1, 2 }	0	0	0	0	0	1	7	75	1,117	14,380
{0, 1, -1}	0	0	0	0	0	1	7	76	1,119	14,382
$[0, 1] \in \mathbb{R}$	0	0	0	0	0	1	7	77	1,120	14,420

Even though the implementation of Strassen's algorithm might seem a little hard to do, if we follow the algorithm as is presented in *Introduction to Algorithms* by CLRS, the implementation is straightforward.
The Table 2 presents the results of running the same experiments, from which we can see that the algorithm performs not too good compared to the standard multiplication algorithm. In fact, the performance is poor if we consider that the difference is 1 or 2 orders of magnitude. The poor performance can be mainly due to 2 reasons:

1. Strassen's algorithm has a good performance with matrices of medium size, since we are dividing the size of the matrices up to 1, we are computing a lot of matrices of the same size (but not the same ones). This can be improved by finding a threshold and run the Strassen's algorithm up to that threshold and then use the standard multiplication algorithm.

2. Memory allocation and deallocation. During the recursive calls, a great amount of sub matrices are allocated and subsequently deallocated. This can also generate a slowness in the performance.

3 Finding a threshold

Based on the Table 1 experimental thresholds can 32 and 64. Analytically the threshold can be computed considering different cases as follows.

Table 2: Strassen's multiplication algorithm run with different test cases and sizes

Test\n	2	4	8	16	32	64	128	256	512	1024
{0, 1}	0	0	2	26	181	366	1,231	8,343	56,602	429,100
{0, 1, 2 }	0	0	1	4	34	219	1,456	8,845	69,767	491,296
{0, 1, -1}	0	0	1	4	30	231	1,489	10,367	69,940	491,382
$[0, 1] \in \mathbb{R}$	0	0	1	4	30	215	1,454	9,969	70,260	490,370

If we consider only the multiplication in the Strassen's algorithm, then we have the recurrence $T(n) = 7 \cdot T(n/2)$ and if we consider $n = 2^k$, then the recurrence becomes $T(2^k) = 7 \cdot T(2^k/2) = 7 \cdot T(2^{k-1})$, which can be seen as $T(k) = 7 \cdot T(k-1)$ and then we can get the characteristic equation as $x - 7 = 0$ and we get that $x = 7$, so the solution to the equation is $T(k) = c_1 \cdot 7^k$. If $n = 2^k$, then applying logarithm function based 2 on both sides, we get that $\log_2 n = \log_2 2^k = k$. Hence we have that $T(n) = c_1 \cdot 7^{\log_2 n} = c_1 n^{\log_2 7}$. If we consider $T(1) = 1$, since the base case would to just return the multiplication of two numbers, then we have that $T(1) = c_1(1)^{\log_2 7} = c_1 = 1$. Therefore with this weak analysis, we conclude that a good threshold would be $n = 1$, and if we naively try with values like 2, 4 and even 8, it might the wrong idea that the threshold is correct.

If we consider the multiplication and addition as operations with the same amount of unit effort, then we can see Strassen's algorithm as a recurrence of using 7 multiplication of matrices of sizes $n/2$ and 18 additions. This will help us find a better threshold.
Based on the above, we have that Strassen's recurrence is given by equation (1).

$$\begin{aligned} T(n) &= 7 \cdot T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 \\ &= 7 \cdot T\left(\frac{n}{2}\right) + 18\left(\frac{n^2}{4}\right) \\ &= 7 \cdot T\left(\frac{n}{2}\right) + \frac{18}{4}\left(n^2\right) \\ &= 7 \cdot T\left(\frac{n}{2}\right) + \frac{9}{2}n^2 \end{aligned} \tag{1}$$

If we consider $n = 2^k$ then the Strassen's recurrence can be seen as in equation (2).

$$\begin{aligned} T(2^k) &= 7 \cdot T\left(\frac{2^k}{2}\right) + \frac{9}{2}\left(2^k\right)^2 \\ &= 7 \cdot T\left(\frac{2^k}{2}\right) + \frac{9}{2}\left(2^{2k}\right) \\ &= 7 \cdot T\left(\frac{2^k}{2}\right) + \frac{9}{2}\left(2^2\right)^k \end{aligned}$$

$$
\begin{aligned}
&= 7 \cdot T\left(\frac{2^k}{2}\right) + \frac{9}{2}(4)^k \\
&= 7 \cdot T\left(2^{k-1}\right) + \frac{9}{2}(4)^k \qquad (2)
\end{aligned}
$$

From equation (2) we can get the characteristic equation shown in equation (3).

$$t_k - 7 \cdot t_{k-1} = 4^k \left(\frac{9}{2}\right) \qquad (3)$$

The characteristic polynomial from equation (3) is presented in equation (4).

$$p(x) = (x-7)\left(x - \frac{9}{2}\right) \qquad (4)$$

The roots of the polynomial in equation (4) are given by $x_1 = 7$ and $x_2 = \frac{9}{2}$, which yields the solution to equation (3) which is shown in equation (5).

$$t_k = c_1 7^k + c_2 \left(\frac{9}{2}\right)^k \qquad (5)$$

Remembering that $n = 2^k$ and then $k = \log_2 n$, we have that the equation (5) can be seen as in equation (6) or as in equation (7).

$$
\begin{aligned}
T(n) &= c_1 7^{\log_2 n} + c_2 \left(\frac{9}{2}\right)^{\log_2 n} \qquad (6) \\
&= c_1 n^{\log_2 7} + c_2 n^{\log_2 9/2} \\
&= c_1 n^{\log_2 7} + c_2 n^{\log_2\left(9 \cdot 2^{-1}\right)} \\
&= c_1 n^{\log_2 7} + c_2 n^{\log_2 9 + \log_2 2^{-1}} \\
&= c_1 n^{\log_2 7} + c_2 n^{\log_2 9 - 1} \\
&= c_1 n^{2.8} + c_2 n^{3.169-1} \\
&= c_1 n^{2.8} + c_2 n^{2.169} \qquad (7)
\end{aligned}
$$

The equation (6) is going to help us find the constants c_1 and c_2 and equation (7) is going to help us find the threshold.
Considering that $T(1) = 1$, then for $T(2)$ the solution is 25 as shown in equation (8).

$$
\begin{aligned}
T(2) &= 7 \cdot T\left(\frac{2}{2}\right) + \frac{9}{2}\left(2^2\right) \\
&= 7 \cdot T(1) + \frac{9}{2}(4) \\
&= 7 + 18 \\
&= 25 \qquad (8)
\end{aligned}
$$

For $T(4)$, we have that the solution is 247, as shown in equation (9).

$$\begin{aligned} T(4) &= 7 \cdot T\left(\frac{4}{2}\right) + \frac{9}{2}\left(4^2\right) \\ &= 7 \cdot T(2) + \frac{9}{2}(16) \\ &= 7(25) + 9(8) \\ &= 175 + 72 \\ &= 247 \end{aligned} \tag{9}$$

Therefore, from equations (8) and (9) and considering the solution in equation (6) we have relationship shown in equation (10).

$$\begin{array}{rcrcl} T(2) &=& c_1 7^{\log_2 2} + c_2 \left(\frac{9}{2}\right)^{\log_2 2} &=& 25 \\ T(2) &=& c_1 7 + c_2 \left(\frac{9}{2}\right) &=& 25 \\ & & & & \\ T(4) &=& c_1 7^{\log_2 4} + c_2 \left(\frac{9}{2}\right)^{\log_2 4} &=& 247 \\ T(4) &=& c_1 7^2 + c_2 \left(\frac{9}{2}\right)^2 &=& 247 \end{array} \tag{10}$$

From the substitution in equation (10), we have the linear system in equation (11).

$$\begin{array}{rcl} 7 \cdot c_1 + \left(\frac{9}{2}\right) c_2 &=& 25 \\ & & \\ 49 \cdot c_1 + \left(\frac{81}{4}\right) c_2 &=& 247 \end{array} \tag{11}$$

The linear equation system can be solved in four steps. The first step is presented in equation (12), in which we multiply by -7 the first row and add it to the second row.Then the second row is divided by -11.25 which yields the value for $c_2 = -6.4$ as shown in equation (13).
For the third step, we just divide by 7 the first row as shown in equation (14) and finally we multiply by $-14/9$ the second row and we add it to the first row as shown in equation (15).

$$\left(\begin{array}{cc|c} 7 & \frac{9}{2} & 25 \\ & & \\ 49 & \frac{81}{4} & 247 \end{array}\right) \sim \left(\begin{array}{cc|c} 7 & \frac{9}{2} & 25 \\ & & \\ 0 & -11.25 & 72 \end{array}\right) \tag{12}$$

$$\left(\begin{array}{cc|c} 7 & \frac{9}{2} & 25 \\ & & \\ 0 & -11.25 & 72 \end{array}\right) \sim \left(\begin{array}{cc|c} 7 & \frac{9}{2} & 25 \\ & & \\ 0 & 1 & -6.4 \end{array}\right) \tag{13}$$

$$\left(\begin{array}{cc|c} 7 & \frac{9}{2} & 25 \\ & & \\ 0 & 1 & -6.4 \end{array}\right) \sim \left(\begin{array}{cc|c} 1 & \frac{9}{14} & 25/7 \\ & & \\ 0 & 1 & -6.4 \end{array}\right) \tag{14}$$

$$\left(\begin{array}{cc|c} 1 & \frac{9}{14} & 25/7 \\ & & \\ 0 & 1 & -6.4 \end{array}\right) \sim \left(\begin{array}{cc|c} 1 & 0 & 13.526 \\ & & \\ 0 & 1 & -6.4 \end{array}\right) \tag{15}$$

From the solutions presented above we have that $c_1 = 13.526$ and $c_2 = -6.4$, which yields the solution presented in equation (16).

$$T(n) = 13.526n^{2.8} - 6.4n^{2.169} \tag{16}$$

Now we can determine when Strassen's algorithm is going to be faster than the standard multiplication algorithm by finding a solution to $13.526n^{2.8} - 6.4n^{2.169} = n^3$, which can be easily programmed using binary search or using some Mathematical program like Sage. In sage, if I try

```
(13.526 * x^2.8 - 6.4 * x^2.169 == x^3).find_root(452000,452600)
```

I get that the solution is $x = 452448.8633404477$, i.e., we would like to compute matrices whose sizes are in the hundreds of thousands to verify that Strassen's algorithm is faster.

Based on the above analysis, the algebraic analysis does not help to find a suitable threshold which would help us compute the multiplication of matrices faster, so for the next experiment, we are going to use the experimental threshold.

4 Strassen with threshold

In this section I am going to present the results of running the naive implementation of Strassen's algorithm using thresholds. The first two threshold that I use are 32 and 64 because those are the values whose running was below 0 and 1 milliseconds, compared to Strassen's values which running time was already around 30 and 200 milliseconds.
The higher the threshold, the better results it is going to give, but the idea of this assignment is to find the minimum threshold. In Table 3, we can see the results of running the same test cases with the same sizes of the matrices and the performance is so much better.

Table 3: Strassen's multiplication algorithm with threshold 32 run with different test cases and sizes

Test\n	2	4	8	16	32	64	128	256	512	1024
{0, 1}	0	0	0	0	2	3	56	98	544	3,565
{0, 1, 2 }	0	0	0	0	0	1	7	60	546	3,106
{0, 1, -1}	0	0	0	0	0	1	16	54	443	2,972
$[0, 1] \in \mathbb{R}$	0	0	0	0	0	5	8	61	432	3,056

The Table 4 presents the results of running the naive Strassen's implementation with a threshold of 64. Based on the previous analysis, the obtained results were expected

Table 4: Strassen's multiplication algorithm with threshold 64 run with different test cases and sizes

Test\n	2	4	8	16	32	64	128	256	512	1024
{0, 1}	0	0	0	1	19	9	68	67	484	3,006
{0, 1, 2 }	0	0	0	0	0	1	6	56	432	2,778
{0, 1, -1}	0	0	0	0	0	1	7	59	454	2,599
$[0, 1] \in \mathbb{R}$	0	0	0	0	0	1	6	54	624	2,578

since as the threshold gets higher (but no higher than 450,000), this method will compute faster than the standard matrix multiplication.

Even though a threshold of 32 might seem to be a good one, we require a the minimum threshold possible which computes the multiplication of 2 matrices faster than the standard multiplication algorithm. The Table 5 presents the results of running the same test cases with the same sizes, but with a threshold of 16. As expected, the performance is not as good as with a threshold of 64 or 32, but still better than the standard matrix multiplication algorithm.

Table 5: Strassen's multiplication algorithm with threshold 16 run with different test cases and sizes

Test\n	2	4	8	16	32	64	128	256	512	1024
{0, 1}	0	0	0	1	12	32	31	157	1,375	5,506
{0, 1, 2 }	0	0	0	0	0	1	11	93	619	4,392
{0, 1, -1}	0	0	0	0	0	1	11	90	616	4,307
$[0, 1] \in \mathbb{R}$	0	0	0	0	0	7	11	91	617	4,305

Based on the above experiments, it might appear that with a threshold of 8, we could still get better results that the standard matrix multiplication algorithm. As can be seen in the Table 6, the threshold of 8 is still good, and it is the one that I am going to use as default.

Table 6: Strassen's multiplication algorithm with threshold 8 run with different test cases and sizes

Test\n	2	4	8	16	32	64	128	256	512	1024
{0, 1}	0	0	0	1	14	30	47	481	1,555	9,161
{0, 1, 2 }	0	0	0	1	1	4	25	184	1,249	8,705
{0, 1, -1}	0	0	0	0	0	1	25	190	1,236	8,737
$[0, 1] \in \mathbb{R}$	0	0	0	0	0	3	23	182	1,249	8,961

5 Working with non power of 2 matrices

Strassen's algorithm works fine for matrices that are square and the size is a power of 2. For the case when the size is not a power of 2, or they might not even be square they algorithm does not divide correctly the sub matrices.
To solve this problem, I use the technique of padding the matrices up to the next power of 2 and the after the multiplication is done, I removed the padding. The Table 7 presents the results of running the same test cases but with different matrices sizes. Note that for these scenarios, the matrices are still square and the running time still in what it is expected to be.

Table 7: Strassen's multiplication algorithm with threshold 8 run with different test cases and sizes

Test\n	3	5	9	17	34	63	125	250	500	1000
{0, 1}	0	0	0	0	4	4	25	182	1,256	8,667
{0, 1, 2 }	0	1	1	1	4	4	128	182	1,255	8,670
{0, 1, -1}	0	1	1	2	5	5	27	182	1,299	8,669
$[0, 1] \in \mathbb{R}$	1	1	1	4	28	11	88	369	1,443	9,081

The problem occurs when we have matrices with sizes equal to $2^k + 1$, because the matrices will have to be padded up to 2^{k+1}. For this scenario, the threshold 8 might not be a good choice, since it is almost running in the same magnitude as the standard matrix multiplication. To try to improve the performance when this scenario happens, the threshold to be chosen is 32 and the results while running these scenarios are presented in Table 8.

Table 8: Strassen's multiplication algorithm with threshold 32 run with different test cases and sizes

Test\n	3	9	17	33	65	129	257	513	1025
{0, 1}	1	1	1	2	10	82	851	3,989	27,761
{0, 1, 2 }	3	1	1	4	10	131	803	3,926	27,620
{0, 1, -1}	3	1	1	4	11	121	851	3,900	27,317
$[0, 1] \in \mathbb{R}$	1	1	27	15	52	203	864	4,855	28,672

The problem here is that we are multiplying a lot of matrices that are 0, so we just need an indicator that if a matrices is 0, then we just return it. Currently, the

6 Conclusion

After finishing this assignment and the experiments, we can conclude that we can make Strassen's algorithm to run faster than standard multiplication algorithm even with matrices of small sizes.

The padding technique can be improved by con siding an adaptive threshold and based on the dimension and the next power of 2, can determine the best threshold for each particular scenario.

There was one solution not presented in this report, in which the division of matrices was done without generating a new matrix. Instead the matrix object took the general matrices and its corresponding size of the sub matrix and using offsets it is able to retrieve the corresponding element of each sub matrix. This saves a lot of memory since we are not creating sub matrices but for the initial runs it was running slower than the naive implementation. Maybe the use of a threshold would have drastically improved the performance of this technique.

The algorithms were implemented using Java SE 1.7 and maven which the current *NICE* system does not support. So everything has been packed and uploaded to my account and can be run using

```
java -jar Strassen.jar 0 3 input.txt
```

This have been tested, so I expected no problems while running it. If the user would like to try it in a different environment or recompile it again from scratch, you would only need to run the following two commands:

```
mvn clean compile assembly:single
mvn assembly:assembly
```

After the command second command is executed, it can be killed when running the test cases.