CS124 Programming Assignment 2
Yihua Leng, Daniel Schwartz
3-22-2021

Introduction

This report concerns the effectiveness of Strassen's Algorithm for matrix multiplication and the conventional matrix multiplication algorithm. The conventional algorithm has a run time of n^3 while Strassen's has a better runtime of $n^{log_2 7}$. But it may not always be necessary to run Strassen's algorithm, a divide and conquer algorithm, all the way to a base case of 1. It may make more sense to run it until it hits a case that can be handled fast by our conventional algorithm. So our first goal was to analytically determine the crossover point in Strassen's algorithm that would optimally switch over to the conventional matrix multiplication algorithm. This is the point at which it is no longer beneficial to keep recursively calling Strassen's algorithm. Our second goal was to implement the conventional algorithm in C++ and also Strassen's Algorithm, tying in a crossover point to the conventional algorithm. These algorithms must work for powers of 2, even, and odd numbers, which adds some complexity. After our implementation, we would be able to experimentally test the most efficient crossover point in our Strassen Algorithm. And finally, we wanted to use the matrix multiplication functions we created to determine the number of triangles in an undirected graph, encoded with an adjacency matrix.

Analytical Analysis:

To get a sense of what we should expect for the crossover point when it comes to the algorithm implementation, it was important to determine the theoretical optimal crossover point in our Strassen Algorithm using the algorithm's recurrence relations. In class, we learned of Strassen's remarkable cut from 8 operations to 7 operations as shown below:

$$P_1 = A(F - H)$$
$$P_2 = (A + B)H$$
$$P_3 = (C + D)E$$
$$P_4 = D(G + E)$$
$$P_5 = (A + D)(E + H)$$

$P_6 = (B - D)(G + H)$
$P_7 = (A - C)(E + F)$

$AE + BG = P_5 + P_4 - P_2 + P_6$
$AF + BH = P_1 + P_2$
$CE + DG = P_3 + P_4$
$CF + DH = P_5 + P_1 - P_3 + P_7$

First we will calculate the crossover value for the optimal when the dimension is a power of 2. This gives us a formula of $S(n) = 7S(n/2) + \Theta(n^2)$, and the $\Theta(n^2)$ comes out to $18(n^2)$because we have 18 addition/subtraction operations. So now we have $S(n) = 7S(n/2) + 18(n^2)$. For the conventional algorithm, we know we $C(n) = n^2(2n - 1)$because it costs $2n - 1$ to compute n^2entries. Our Strassen's algorithm is optimal when it crosses over to the conventional algorithm at the point when the conventional algorithm is faster than Strassen's algorithm. So what we are really doing in the algorithm is taking the minimum among our our conventional and Strassen each time such that $S(n) = 7[MIN(S(n/2), C(n/2))] + 18(n^2)$. So when $C(n/2)$is the minimum value, we will have the value of our crossover. Mathematically, we will have our crossover when $n = 15$by the calculations below.

$C(n) = 7C(n/2) + 18(n^2)$
$n^2(n - 1) = (7n^2/4)(n - 1) + 9n^2/2$
$n = 15$

Implementation

We first needed a matrix struct that is composed of a number of rows, number of columns, a matrix, and a padding integer. We will discuss more about the padding integer later.

We then implemented the conventional algorithm for matrix multiplication because we knew we would need it for our Strassen's algorithm. This implementation was fairly straightforward, running in $O(n^3)$ time because we have to loop through matrices 3 times for each entry.

Next, we implemented our Strassens function that takes a matrix with dimensions that are a power of 2. This was important because when we recursively split our matrix for the algorithm, we will run into errors as the quadrant dimensions get low and the dimensions aren't a power of 2. Our split function takes a matrix and splits it into 4 equal dimension submatrices.

But this raised another problem as well. What do we do when we want to calculate an odd matrix or an even matrix who's dimensions aren't a power of 2? To overcome this hurdle, we first decided to make a padding function that would pad our matrix with zeros up to the next power of 2 for each dimension. This doesn't affect our end calculation because padding with zeros only results in the final answer having padding as well. It doesn't affect the values that we care about in our final matrix. And to determine how much padding we used, we created a function called compute_pad that checked the padding to the newest power of 2.

This padding method works, but we realize it wasn't the most efficient. In reality, we only need the dimension to be divisible by 2 until we hit our crossover value, because once we hit our crossover value and use our conventional algorithm, the dimensions don't need to be powers of 2 or even be divisible by 2. So we take our crossover value and double it until it is just bigger than the matrix dimension. Then when we divide and conquer, we don't need to go until 1. We just need to go until our crossover value.

Putting this all together, we recursively called the operations written above to finish this divide and conquer algorithm. Of course we added helper functions to add and subtract matrices as well.

We also created a function strassen_pad that computes the padding based on the dimensions of the matrices and runs Strassens on the padded matrices, returning a correct matrix. The strassens_pad function and our conventional function are our resulting functions that we use to run our experiments.

Testing

We first manually imputed matrices that we could calculate ourselves and ran the conventional algorithm. We imputed dimensions that were powers of 2, other even dimensions, and odd dimensions. Once we tested the conventional algorithm sufficiently to show its correctness, we used the result from the conventional algorithm to test our Strassen algorithm. We created a random matrix generator function that generated a

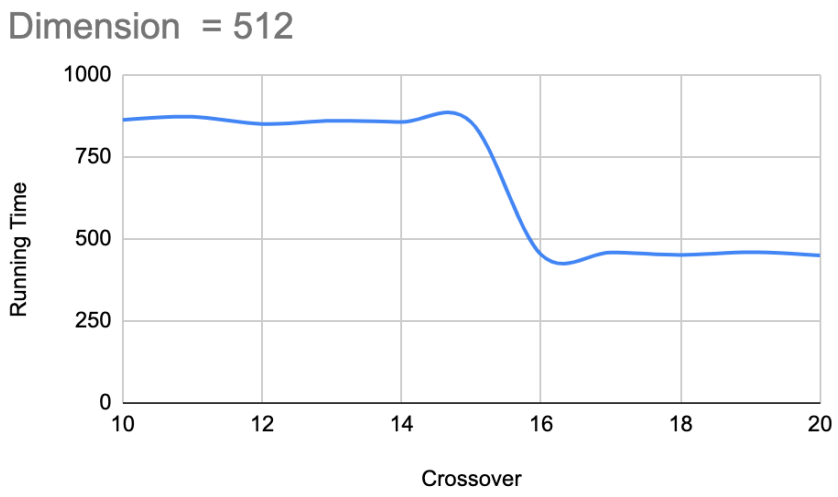
random matrix of zeros and ones, given two dimensions. To test our Strassens algorithm, we generated two random matrices with this function that fit the constraints of matrix multiplication, produced an expected answer with our conventional algorithm, then we ran strassen's algorithm to test whether or not our strassen's result matched the expected (conventional) result.

Crossover results

To run our experiment we wanted to see what the optimal crossover values were for various dimensions, so we timed 5 trials of our Strassen's algorithm running on a single dimension with multiple crossover values from 0 to dimension. The goal of this was to identify the dip or "valley" which is the point where a certain crossover value results in the fastest runtime.

For powers of 2, there is always a sharp drop off in runtime once we reach the crossover value of 16. This valley is close to our theoretical crossover of 15 for our trials on dimensions of 256, 512, and 1024.

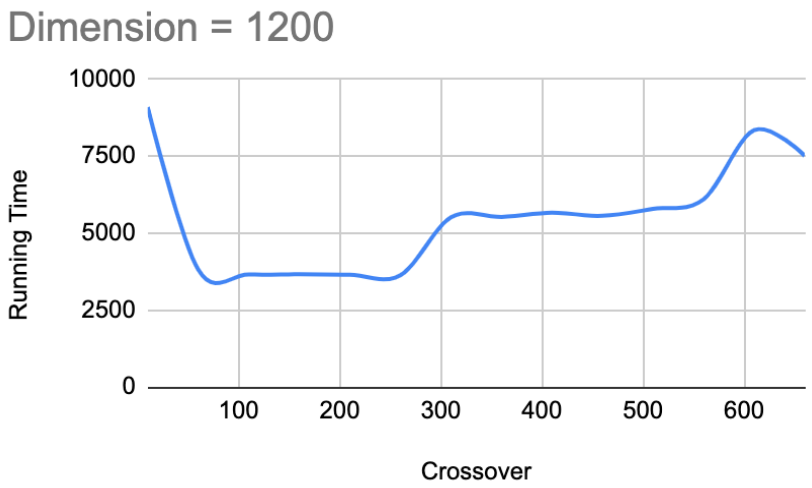
	Dim=256	Dim=512	Dim=1024
Crossover	Time	Time	Time
10	122.769473	862.756162	6861.486700
11	141.733423	872.266125	6647.479819
12	126.131928	850.313072	6427.432623
13	125.679621	860.102732	6522.987006
14	125.981031	856.469278	6447.308444
15	125.087582	855.493176	6356.339746
16	66.963933	451.702673	3416.549765
17	67.283573	458.636855	3284.901087
18	66.664010	451.015734	3215.375957
19	67.230224	459.261379	3184.197421
20	67.354045	449.443445	3247.413074



For non-powers of 2 we typically get a valley around 40, which you can see in the charts below. When the dimension is 600, there is a large drop off in time when we make the crossover somewhere around 40-50, which is apparent in the graph. Similarly, for dimensions 800, 1000, and 1200, there is typically a drop off in time somewhere between 60 and 110.

D=600		D=800	
crossover: 10	time: 1590.525859	crossover: 10	time: 5989.740312
crossover: 60	time: 729.237988	crossover: 60	time: 1522.756544
crossover: 110	time: 656.835955	crossover: 110	time: 1492.702893
crossover: 160	time: 676.526402	crossover: 160	time: 1487.372188
crossover: 210	time: 676.120500	crossover: 210	time: 1580.842532
crossover: 260	time: 675.345116	crossover: 260	time: 1584.382775
crossover: 310	time: 824.836038	crossover: 310	time: 1607.299187
crossover: 360	time: 816.379477	crossover: 360	time: 1573.044292
crossover: 410	time: 816.980531	crossover: 410	time: 2235.488595
crossover: 460	time: 815.866309	crossover: 460	time: 2227.477891
crossover: 510	time: 815.748961	crossover: 510	time: 2221.846728
crossover: 560	time: 816.162412	crossover: 560	time: 2188.309860

D=1000		D=1200	
crossover: 10	time: 7483.519946	crossover: 10	time: 9085.206960
crossover: 60	time: 3223.249380	crossover: 60	time: 3806.070001
crossover: 110	time: 2839.390109	crossover: 110	time: 3653.861949
crossover: 160	time: 2937.009522	crossover: 160	time: 3656.470402
crossover: 210	time: 2932.846904	crossover: 210	time: 3643.326593
crossover: 260	time: 3051.095605	crossover: 260	time: 3624.325610
crossover: 310	time: 3056.760971	crossover: 310	time: 5506.673942
crossover: 360	time: 3041.205634	crossover: 360	time: 5520.222499
crossover: 410	time: 3054.915485	crossover: 410	time: 5655.335697
crossover: 460	time: 3088.129732	crossover: 460	time: 5555.188014
crossover: 510	time: 4482.310229	crossover: 510	time: 5781.792575
crossover: 560	time: 4500.820651	crossover: 560	time: 6091.123425



To summarize these results, it is clear that the optimal crossover for powers of 2 typically ranges from 16 to 19, while the optimal crossover for other numbers ranges from the mid 30s to 110. This is a broad range, so we tried a few crossover values within a more narrow range. Zooming in, we see that a good estimate of optimal crossover for non-powers of 2 is around 36-40: the crossover values on dimension 600 at 36 and 37 are in the mid-700s while the crossover values on 38 and 39 are in the low 500s. This pattern is sustained for other dimensions.

Adjacency Matrix Analysis

An interesting use case for our new algorithm is in determining the number of triangles in an undirected graph, which can be represented as an adjacency matrix, where a 1 represents an edge from vertex i to vertex j and a 0 represents no edge. We wanted to build a matrix that randomly populated each ijth entry in a 1024 X 1024 matrix with a zero or one, given a probability p for a 1 to be in that entry. Since the graph is undirected, we know the matrix will be symmetrical since for every ij that has an edge, there is also an edge ji. Therefore, it is important to populate the jith entry with the same value that we populate the ijth entry. Along those same lines, we don't want to assign our ji entry a value with probability p if we already assigned ija value with probability p. It is only necessary to assign one "direction" of each edge. So we simply randomly populated the upper right corner of the matrix and then mirror each value to the other side.

Once we had this representation of a graph, we could determine the number of triangles by cubing the adjacency matrix, summing the diagonal of the resulting matrix, and dividing this value by 6. We divide by 6 because each vertex in a triangle will be counted as one triangle. And in our situation of an undirected graph, each vertex will be counted as two triangles. So we are over counting a triangle as 6 triangles rather than 1.

We ran the program on p values of .01, .02, .03, .04, .05, and we can compare our values (number of triangles) to the expected values using the equation $1024C3 \times p^3$. The below table shows our values compared to the expected values.

Probability	Strassen	Expected
0.01	181	178.433014
0.02	1431	1427.464111
0.03	4848	4817.691406
0.04	11444	11419.71289
0.05	22404	22304.12891

Discussion

We noticed some discrepancies between our predicted crossover values and our calculated values. One reason for these discrepancies could be that in our theoretical analysis, we just accounted for mathematical operations like addition and subtraction. Surely these operations are the most time consuming part of our algorithm, but perhaps things like memory allocation also have a cost that can affect our results. To pile onto this, some crossover values cause an allocation of more memory than one would think because of our matrix padding. So, for example, with a crossover value of 1, we need to pad our matrix to the power of 2 after our dimension. So a cross over value of 1 and a dimension of 1025 would lead us to allocate memory for a 2048 X 2048 matrix rather than a 1025 matrix.

One thing we could have done to make our findings more accurate is run the program for specific cutoff values, rather than broad ranges, for dimensions 600, 800, 1000, 1200. The reason we didn't was because when running the program for every cutoff value 0 through dim, the running time becomes exponentially large.

If we were making this program for long term use where we want to consistently find the most optimal cutoff value given two matrices with random dimensions, it may be beneficial to make a lookup table that returns the optimal cutoff value given a dimension. This would save time, as we wouldn't have to test multiple crossover values whenever we are given a new matrix.

This turned out to be a fascinating project, despite the idea of matrix multiplication initially seeming monotonous and dull. The reason the project turned out to be fascinating was because I realized, as I often do in this class, that ostensibly unchanging solutions to problems can almost always be improved. Sometimes it takes a magnificent mind like Volker Strassen's, but the idea of constant improvement leaves me optimistic.