Programming Assignment 2

Rita Bogdanova-Shapkina

April 2, 2022

Late days used on this pset: 2
Total late days used: 8

1 Algorithm Analysis

Given two $n \times n$ matrices X and Y, their product is defined by the formula

$$XY = Z, \quad z_{ij} = \sum_{k=0}^{n-1} x_{ik} y_{ki}$$

where z_{ij} is the element in the ith row and jth column of Z (and respectively likewise for x_{ij} and y_{ij}). Therefore, the basic matrix multiplication algorithm, which iterates over all pairs of (i, j) and for each pair multiplies $x_{ik} \cdot y_{ij}$ from $k = 0$ to $k = n - 1$, adding all sums, has a running time $b(n) = n^2(2n - 1)$. This is because all n^2 pairs (i, j) are iterated over, and for each pair n multiplications (the $x_{ik} \cdot y_{ij}$) and $n - 1$ additions are performed (when all operations besides arithmetic are "free," the algorithm is most efficient if when $k = 0$, $x_{i0} \cdot y_{i0}$ is not added to any existing sum and instead z_{ij} is simply set to $x_{i0} \cdot y_{i0}$ - all $n - 1$ following $x_{ik} \cdot y_{ik}$ are then added to z_{ij}). Thus, the runtime of the basic algorithm when integer addition and multiplication have a cost of 1 and all other operations are 'free' is $b(n) = 2n^3 - n^2$.

Alternatively, the Strassen algorithm takes advantage of the fact that the product of two matrices (in block matrix form) can be expressed as

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

where the matrices $A, \ldots, H$ are square of side length $\frac{n}{2}$, with

$$Q1 = AE + BG = P4 + P5 + P6 - P2$$

$$Q2 = AF + BH = P1 + P2$$

$$Q3 = CE + DG = P3 + P4$$

$$Q4 = CF + DH = P1 + P5 + P7 - P3$$

and

$$P1 = A(F - H)$$

$$P2 = (A + B)H$$

$$P3 = (C + D)E$$

$$P4 = D(G - E)$$
$$P5 = (A + D)(E + H)$$
$$P6 = (B - D)(G + H)$$
$$P7 = (C - A)(E + F)$$

Note that under the the current time cost framework, adding / subtracting two square matrices of side length l takes exactly l^2 time, as one addition / subtraction is performed between each corresponding pairs of entries in the two matrices. Then, calculating all of the Q_is requires 8 additions / subtractions of $\frac{n}{2}$-side-length square matrices (the P_j), and calculating the $P_j$requires 10 additions / subtractions, and 7 multiplications, of $\frac{n}{2}$-side-length square matrices. Thus, the runtime of Strassen's algorithm (in isolation) can be described by the recurrence relation $s(n) = 7s(\frac{n}{2}) + 18(\frac{n}{2})^2$, $s(1) = 1$ (since multiplying 2 matrices of size 1 is just an integer multiplication).

An improved algorithm will use Strassen's algorithm up until its recursion reaches matrices of some threshold size n_0, at which point it will switch to using the basic algorithm. The optimal crossover point n_0 is the matrix size at which it becomes faster to multiply two $n_0 \times n_0$ matrices via the basic algorithm, than to wait until the next recursion of Strassen's algorithm to switch over to basic. So, n_0 is the point such that $b(n_0) = 7b(n_0/2) + 18(\frac{n_0}{2})^2$ - the $7b(n_0/2)$ term represents the 7 multiplications performed by Strassen's, done via the Basic algorithm at the lower level of recursion, and the $18(\frac{n_0}{2})^2$ term represents the additions performed by Strassen's algorithm. Solving for n_0, we have

$$\begin{aligned} b(n_0) &= 7b(n_0/2) + 18(\frac{n_0}{2})^2 \\ 2n_0^3 - n_0^2 &= 7(2(\frac{n_0}{2})^3 - (\frac{n_0}{2})^2) + 18(\frac{n_0}{2})^2 \\ 2n_0^3 - n_0^2 &= 7(\frac{n_0^3}{4}) + 11(\frac{n_0^2}{4}) \\ 8n_0 - 4 &= 7n_0 + 11 \\ n_0 &= 15 \end{aligned}$$

So, the expected cutoff optimal n_0 is 15, if the only operations that take time (and indeed take constant time 1 each) are real-number arithmetic.

Note that this analysis assumes that the matrices (initially) set as input to Strassen's algorithm have a dimension of a power of 2. If matrices with a different dimension than 2^k for some k are multiplied, these calculations over-estimate the efficiency of the combined algorithm by various degrees. There are two ways of padding a non-power-of-two-sized matrix with 0s such that Strassen's can successfully recurse (which will be discussed in detail in the "Implementation" section): initially copying the input matrices (of dimension $n \times n$) to the upper-right corner of a 0 matrix of size $d \times d$, where d is the smallest power of 2 greater than or equal to n, or checking, at every level of recusion, if the dimension of the input matrices is odd and adding 1 row and column of 0s if yes. If the former method is used, the same calculation holds except taking the initialize size of the input n to be d. Note that $d - n \leq n - 1$ (if n is 1 greater than a power of 2). If the latter method is used, a greater efficiency for Strassen's may be achieved: though if Strassen's algorithm recurses down to the dimension = 1 level, over each recursion exactly $d - n$ rows / columns will have been added (since the input dimension is checked for even-ness at each level), if at some level of recursion (for dimension > 1) the algorithm switches to Basic multiplication, additional rows are not added even if the matrices being multiplied have (or will have, once divided by 2 enough times) odd dimension, thus resulting in an overall smaller padding than the "pad at once" strategy. How many of the $d - n$ padding rows/columns end up not being added of course depends on the choice of threshold n_0.

2 Implementation

The program was written in C++, to take advantage of the quick compile/execution times of C and C++.

Matrices were represented as two-dimensional vectors to take advantage of vectors' automatic memory deallocation, which is especially useful for recursive algorithms. First, several helper functions were implemented, including testing-helpers for generating random matrices, printing matrices, and timing multiplications of random matrices of a given size, but also the functions `addMat` and `subtractMat` to subtract and add two $n \times n$ matrices. The basic multiplication algorithm in `basicMult` was implemented much as described in the Algorithm Analysis section: looping over all i, j and adding all products $x_{ik}y_{ki}$ for $k \in [n-1]$. To optimize `basicMult` as well as `addMat` and `subtractMat`, they were redefined from functions that return matrices to `void` functions that modify an already-allocated matrix (passed by reference as an additional function parameter), and the input matrices were also passed by reference; this reduced the times memory allocation was needed for the creation of copies (and allowed further optimization of the combined-Strassen's algorithm - see below) of matrices, and led to a $\sim$ 2x improvement in the performance of `basicMult` for n between 100 and 4100, as shown by testing.

Strassen's algorithm was also implemented similarly to its description in the Algorithm Analysis section: 8 2D vectors of dimension $n/2$ are allocated and set equal to the suqarter-divisions of the input matrices, A through H; then, 7 2D vectors of dimension $n/2$ are allocated to store $P1, \ldots, P7$ as described in the first section and their contents are calculated via the formulas above, via the use of `addMat` and `subtractMat` and recursive calls to Strassen's algorithm. Then, 4

To create the combined Strassen's/basic algorithm, the base case recursion from Strassen's algorithm was changed from the $n = 1$ case (direct multiplication) to the $n \leq n_0$ case, where n_0 is a given threshold (altered during testing - see below) at which the multiplication is performed via a call to the Basic algorithm.

The first optimization that was implemented was the passing of the two matrices to be multiplied, as well as a pointer to a pre-allocated product matrix, by reference to the algorithm, which eliminated the need to allocate memory for new matries (for the result and copies of the input) at every level of recursion. The second optimization implemented was to, instead of allocating memory for a new matrix for the result of the additions / subtractions needed to calculate the P_j, only two additional matrices, `temp1` and `temp2` were created. Since `addMat` and `subtractMat` make in-place modifications, `temp1` (and `temp2`, as needed when two additions are needed to calculate the same P_j) is made to hold the intermediate sum/difference, then is fed as an input to a recursive iteration of Strassen's (for the multiplication), and then is re-used to calculate the next P_j. For example, to find $P4 = D(G - E)$, first $G - E$ is stored in `temp1`, then `temp1` and D are multiplied and stored in $P4$. Then, to find $P5 = (A + D)(E + H)$, first $A + D$ is stored in `temp1`, then $E + H$ is stored in `temp2`, then `temp2` and `temp1` are multiplied and stored in $P5$, and so on. This eliminated the need to allocate new matrices for each intermediate sum and difference. The final optimization was not to allocate memory for new matrices for $Q1, Q2, Q3, Q4$ but instead modify the product matrix directly from the P_js, using an iteration from 0 to $(n/2) - 1$ and the formulas for each Q_i, adding $n/2$ to the row and/or column index of the element a $P_j[i][j]$ is added to, depending on which 'quadrant' of the resultant product matrix is targeted.

Initially, this algorithm only worked at a given level of recursion if the current input matrix dimension is even, and so only worked overall if the input matrices' dimension was a power of 2. As explained at the bottom of the Algorithm Analysis section, there were two options available for padding the input matrices with 0s - either up-front padding to a size of $d \times d$, where d is the smallest power of 2 greater than (or equal to, but then there is no padding done or needed) n, or by-level padding - with the latter method theoretically being more efficient as not all $(d - n)$ rows/columns of padding will be added due to the crossover to basic multiplication (and thus no more padding additions even if the matrix dimension is odd). However, in practice, this would be difficult to implement without increasing the runtime through memory allocation for new matrices during every recursion (though may be possible by avoiding new matrices altogether and using pointer arithmetic). So, up-front padding was implemented by pre-processing the input to the combined Strassen's algorithm to be two $d \times d$ array where all elements outside the original matrices are equal to 0 (and the input dimension passed is d, not n).

Since in the worst case this would almost-double the size of the input matrices (i.e. if $n = 2^k + 1$ for some k, d will be $2^{k+1} = 2n - 1$), So, as an optimization, a `padstart` parameter was passed to Strassen's algorithm, a well as the 'starting' (i.e. top) row and column of the current input matrices relative to the originals. Then, before any operations are performed, it is checked as to whether both the starting row and column of (at least one of) the input matrices is greater than `padstart` - if so, then that matrix is completely within the padding region and so is all 0s, so any product involving it is all 0. So, if one of these checks passes, the result matrix is set to the zero matrix, without further calls to any multiplication algorithms. To pass down correct start row/column values to the recursive multiplication calls, each call's new row/column values are set individually based on a sub-matrix's position relative to its 'parent' matrix (i.e. H has a start row/colun identical to that of its parent MAT1, while H has both its start row and column equal to those of MAT2 plus $n_r/2$, where n_r is the dimension in the current recursion), with values for the sums of matrices set to the minimum between the two (i.e. if H is in the padding region but E is not, $(E + H)$ must be treated as a nonzero matrix). For example, in recursively multiplying $(A + B)H$, the input to Strassen's algorithm is . . . ,, new currRow1 = currRow1, new currCol1 = currCol1, new currRow2 = CurrRow2 + $n/2$, and new currCol2 = currCol2 + $n/2$. The same `padStart` is passed in every recursion, as it is relative to the original input, and the original top-level call for Strassen's has 0 as its row/column start points, and $n + 1$ as its first padded row/column (where d is the size of the padded matrix, as before). This is a useful optimization for matrices that have high amounts of padding, i.e. matrices of original size close to $2^k + 1$ for some k, as it avoids (possibly repeated) multiplication of 0 matrices, and therefore removes much of the downside of 'upfront' padding.

3 Results: Optimal Threshold

Testing the optimal threshold was performed by running the combined Strassen's algorithm on 5 random matrices (with entries from 0 to 2, inclusive - there was no difference for integer ranges up to 10,000, likely because C++ bignum integer handling was not required, but even then testing extremely large integers makes integer multiplication, not matrix multiplication, a larger bottleneck and so does not do much to inform the optimal threshold) and taking the average for each dimension. Dimensions that are powers of 2 - 512, 1024, 2048, and 4096 - were tested, as due to the upfront padding, multiplying matrices of other dimensions, i.e. $2^k < n < 2^{k+1}$ for some k, resulted in runtimes similar to those of multiplying matrices of dimensions 2^{k+1} (or, due to the 0-padding optimization outlined above, runtimes close to those of 2^k-sized matrices if n is close to 2^k). If by-recursion padding has been used instead, it would have been useful to experiment with matrix sizes outside the powers of 2.

The chart outlining results is below, with matrix dimensions accross the top, n_0 inclusive (i.e. matrices of size $\leq n_0$ were sent to the Basic algorithm), and all times in seconds, rounded to the nearest hundredth:

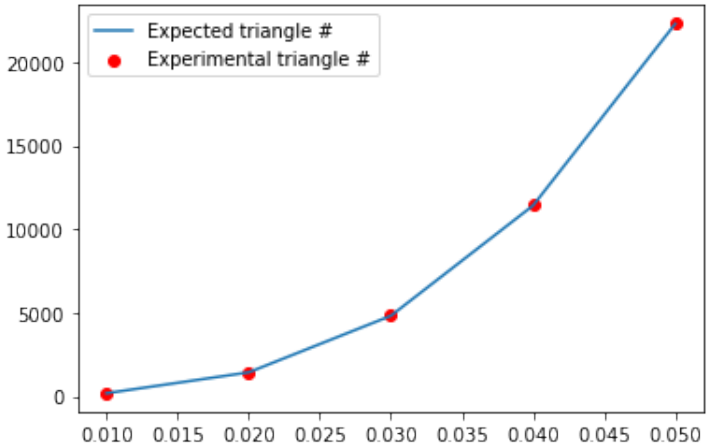
n_0	512	1024	2048	4096
8	1.79	22.66	163.33	735.45
16	1.52	13.57	94.44	622.00
32	1.44	11.51	78.37	523.88
64	1.56	10.97	79.08	549.73
128	1.78	12.11	82.11	586.83
256	2.04	13.05	94.25	658.55
512	2.20	17.74	170.33	781.71

So, experimentally optimal n_0 was determined to be either 32 or 64, with little difference between them. Note that as this implementation involved upfront padding, all entries fed to Strassen's recursive algorithm have a dimension that's a power of 2, and so having a threshold between 32 and 64 is equivalent to having the threshold at 32. If by-recursion padding had been used, it's possible the optimal threshold would have fallen between 32 and 64. In either case, the experimentally optimal n_0 lies significantly above the estimated optimal n_0 of 15, likely due to the unaccounted-for time costs of the many memory allocations in Strassen's algorithm making it less efficient for lower n than theorized.

As expected due to the fact that Strassen's algorithm is more efficient than the basic algorithm for higher n, runtimes increase for n both above and below the optimal n_0: far above n_0 and the basic algorithm is being used for large n_0 where Strassen's would have been more efficient, and for small n_0 much of the benefit of the switch is lost, and Strassen's is being performed for small n where Basic would be more efficient. Notably, the increase in runtimes is gradual for n near n_0, and then sharply rises for n far from n_0 (both above and below), in accordance with the $O(n^{\sim 3})$ nature of both algorithms.

4 Results: Graph Triangles

Graph adjacency matrices were simulated by first creating a matrix (2D vector) of side length 1024 of all 0s, then, iterating through all entries, generated a random number r between 1 and 0 and setting that entry to 1 if $r < p$. Then, that matrix's cube was taken via repeated application of the combined Strassen's algorithm, and the number of triangles was found by summing diagonal entries of the result and dividing by 6. Testing was performed by calling the above function, `GraphCount`, 5 times for each value of p. Below is a graph of the results.



The experimental results lined up closely with the expected results.

Exact results are as follows:

p	Expected	Experimental
0.01	178.433	182.222
0.02	1427.4642	1445.56
0.03	4817.691648	4854.11
0.04	11419.714	11438.003
0.05	22304.128	22329.2

Where the Expected results are based on the formula $E(p) = \binom{1024}{3}p^3$.