CS124 Programming Assignment 2

Liya Jin

Due March 30, 2022

No. of late days used on previous psets: 0

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

1 Introduction

I implemented a modified version of Strassen's algorithm as detailed in the prompt, experimentally determining a runtime-optimal crossover point (n_0) at which I switched from multiplying via Strassen's algorithm to using standard matrix multiplication. I found that the analytically optimal n_0 (assuming constant time for arithmetic calculation) was about 15; I found that the experimentally optimal n_0 was about 11.

2 Finding analytical n_0

Strassen's recurrence: Referencing the Lecture 10 class notes, we note that for every n by n matrix multiplication, by Strassen's algorithm there are:

- 7 matrix multiplications, each between matrices of dimension $\frac{n}{2}$.
- 18 matrix additions and subtractions of matrices of the same dimension, each containing $(\frac{n}{2})^2$ elements.

This gives us the following recurrence equation:

$$T_1(n) = 7T_1(\frac{n}{2}) + 18(\frac{n}{2})^2$$

with the base case $T_1(1) = 1$.

Standard matrix multiplication: For every n by n matrix multiplication, by the standard algorithm there are:

- n^3 total multiplications.
- $n^2(n-1)$ total additions.

We see that this gives us the following runtime equation (closed-form):

$$T_2(n) = (2n-1)n^2$$

Our Strassen's: Since our algorithm combines Strassen's with standard matrix multiplication, we seek to combine these two runtime equations via the cross-over point n_0 as detailed in the programming assignment prompt: letting n be the dimension of the current submatrix, for $n > n_0$ we run Strassens and for $n \leq n_0$ we run the standard multiplication. We then have the equation:

$$T(n) = 7 * \min(T_1(\frac{n}{2}), T_2(\frac{n}{2})) + 18(\frac{n}{2})^2$$

(Notably, the recurrence for T_1 is only accurate when n is a power of 2.) Given our definition of n_0, we also know that:

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

$$\min(T_1(\frac{n_0}{2}), T_2(\frac{n_0}{2})) = T_2(\frac{n_0}{2})$$

We can then plug this back into our joint equation to solve for n_0:

$$T_1(n_0) = 7T_2(\frac{n_0}{2}) + 18(\frac{n_0}{2})^2 = T_2(n_0)$$

which, when solved, reduces to $n_0 = 15$.

3 Standard matrix multiplication

I implemented the standard matrix multiplication algorithm as `reg_mult`, which is given by the following formula (with A and B as factors for product matrix C):

$$C[i][j] = \sum_k A[i][k] * B[i][k]$$

Intuitively, this is most easily implemented with three nested for-loops, one for each iterative element (i, j, k). There are 6 possible orderings of the loops i, j, k, so I tested each experimentally to see whether one was better than the other for $n = 1000$. However, there was no significant difference ($\approx 3\%$ margins) between the different orderings, so I inconsequentially chose the order i, j, k which performed slightly more efficiently.

4 Strassen's (modified) matrix multiplication

4.1 Basic implementation of modified Strassen's

Using Python, I experimented with lists before choosing to use `numpy` to make indexing into matrices as easy as possible. I initially wrote additional functions for matrix addition and subtraction before switching partway to simply using `np.add()` and `np.subtract()`, per Ed post 690. The primary Strassen's function, `my_strassens`, recursively calls itself as it makes matrix quadrant calculations from its two initial inputs until it recurses down to a base case, at which point it switches to regular multiplication.

Using a single-source file made running and testing simple: I wrote flags that ran modified Strassen's, regular multiplication, and calculated triangles.

I also wrote Python scripts to generate test files for matrix multiplication (`mtx_gen.py`), check for output correctness (e.g. `count1s.py`), to generate A^3 matrices for part 3, as discussed later. `mtx_gen.py` in particular creates matrices where each entry was randomly selected to be $0, 1$, or 2 in the same format as the CS124-formatted input file (via Ed post 566).

4.2 Optimizations

4.2.1 Avoiding excessive memory allocation / data copying

The most naive way to implement Strassens's would be to define a new `numpy` array for each of the 8 $\frac{n}{2}$ by $\frac{n}{2}$ subarrays A through G, the 7 intermediate values $P_1, \ldots, P_7$, and each quadrant of the final array. This requires allocating an unnecessary amount of excess memory, resulting in lengthy and unwieldly runtimes.

My first attempt at avoiding naivety was to compute each of the four $\frac{n}{2}$ by $\frac{n}{2}$ submatrices of the result matrix on one line, defining no new memory and simply indexing into the matrix. While this saved much memory, it also cost too much computational power to index into the input matrices that frequently.

I also tried defining variables for each of the input's 8 $\frac{n}{2}$ by $\frac{n}{2}$ subarrays A through G but indexing for $P_1, \ldots, P_7$, and vice versa, but the runtimes were still abysmal.

Ultimately, I tried to make the best tradeoff between the crystal clear organization of the naive approach and the definition of as few variables as possible (thereby requiring less memory and reducing runtime) as follows:

- For each call to `my_strassens`, I define one $\frac{n}{2}$ by $\frac{n}{2}$ intermediate subarray (instead of 15), p_temp, and one matrix for the result matrix, $\texttt{result}_mtx$.
- `p_temp` stores the result of one of the 7 intermediate values $P_1, \ldots, P_7$ at a time, then is directly added / subtracted to the respective quadrant of the result matrix.

- The 8 $\frac{n}{2}$ by $\frac{n}{2}$ submatrices of the inputs never need to be allocated to new memory at all: I index directly into the original n by n matrices as needed when defining the inputs to recursive calls of `my_strassens`.
- While Python is not known for its rigorous memory allocation and deallocation capacities, I used the garbage collector `gc` module to deallocate `p_temp` as soon as possible.

4.2.2 Beyond matrix dimensions of powers of 2

To ensure that the code would work for n by n matrices of any integer n (not just for powers of 2 or even just even matrices), I added a condition within `my_strassens` that checked whether the current matrices' dimensions were even or odd. If so, `my_strassens` is called again with the same inputs padded with an extra row and column of zeros to get the dimensions to an even number: when the result recurses back up, only the non-zero-padded original submatrix's results are returned.

I initially implemented this function to pad up to the next lowest power of 2, but experimentally determined that the runtime was drastically inferior, taking up too much memory and space for large n. Padding just up to the next even number allows for much tighter memory allocation at the cost of number of smaller operations (zero-paddings) for an overall better runtime.

4.3 Testing

With the Python scripts I wrote, I generated class-formatted matrices for small values of n, then plugged them into external (online) matrix multiplication calculators to check. For larger values of n, I used known matrices (such as the identity matrix) to confirm correct multiplication. I discuss my methodology for calculating the crossover point more in-depth below.

5 Results

5.1 Finding experimental n_0

As n_0 gets smaller, the runtime for modified Strassen's initially decreases with respect to the runtime for standard matrix multiplication. However, there is a point at which the former starts increasing back up to the latter: this is the point at which it is no longer more optimal to use the modified version of Strassen's, and we strive to find the smallest n_0 for which modified Strassen's runtime is as close as possible to that of the standard matrix multiplication (once past the initial runtime valley for Strassen's).

To narrow my search, I started by running the algorithm with 1024 by 1024 matrices and checking each potential crossover point, n_0 at all powers of 2. Since I knew working with powers of 2 involved the least runtime "noise" from the extra operations that came with dealing with odd inputs, I used these input

sizes to first narrow down my search. See results in Table 1 below: each entry for each input size and dimension is the average of 5 trials.

Experimenting with n_0 at $n = 1024$		
n_0	`reg_mult`	`my_strassens`
512	742.0308	677.9550
256	742.0308	579.4619
128	742.0308	519.0926
64	742.0308	452.6255
32	742.0308	432.5172
16	742.0308	607.8472
8	742.0308	2120.7860

Table 1: Modified Strassen's algorithm runtimes (in seconds) for $n = 1024$, n_0 only powers of 2.

Notably, Strassen's runtime is lower than that of standard matrix multiplication until it starts to increase between $n_0 = 32$ and $n_0 = 16$, and drastically increases between $n_0 = 16$ and $n_0 = 8$ to bound within it the standard matrix multiplication runtime. We know, then, that we want to explore the values of n_0 between 8 and 16: this is where our odd dimension enhancements come in handy.

Below, I then explored values of n_0 one at a time from $n_0 = 15$ downwards, testing with randomly-generated matrices of dimension $n_0 * 2^6$ (or with 2 raised to a higher power), a size that guarantees that the specific value of n_0 that denotes the crossover point will actually be reached by the program (as opposed to "stepped over", if we start with matrices with power-of-two dimensions).

Experimenting with n_0 from 11-15			
n_0	$n \geq n_0 * 2^6$	`reg_mult`	`my_strassens`
15	960	646.6079	525.1006
14	896	511.9658	443.2100
13	832	429.6035	378.5820
12	768	345.8241	328.0625
11	704	251.8805	261.3098

Table 2: Modified Strassen's algorithm runtimes (in seconds) for $n = 1024$, n_0 only powers of 2.

The crossover point between Strassen's and regular multiplication in Table 2 lies between $n_0 = 11$ and $n_0 = 12$: we conclude that our experimental $n_0 \approx 11$.

5.2 Calculating triangles

I wrote another a Python script (`graph_gen.py`) to generate random graphs of dimensions 1024 by 1024 with probability of edge weight inclusion p. I took

advantage of Python's random module to simulate an edge inclusion probability p, and accounted for the fact that these matrices were unweighted by only running this edge inclusion randomization over the entries above the top left to bottom right diagonal: I left the diagonal out in making the assumption that unweighted matrices do not have self-looped edges (i.e., e from v_2 to v_2). Implementation was a simple matter of including an extra flag and the modifying outputs within `strassen.py`. Each entry below for each input size and each p is the average of 5 trials. Via the assignment prompt, the number of triangles in a graph represented by adjacency matrix A is found by summing the diagonal of A^3 then dividing by 6.

Expected v. Calculated Triangles for $n = 1024$		
p	$\binom{1024}{3}p^3$	Calculated value (from A^3)
0.01	178.433	173
0.02	1427.4642	1449
0.03	4817.6916	4760
0.04	11419.7135	11029
0.05	22304.128	22008
0.06	38541.5332	37647

Table 3: As determined by my modified Strassen's, the calculated value of the number of triangles found in random-generated graphs of $n = 1024$, where each edge was included with various p.

As shown above, the calculated number of triangles for each value of p was within 2.5% of the expected value, $\binom{1024}{3}p^3$. With one exception, the experimental values were slightly less than the expected.

6 Discussion

6.1 High runtimes

6.1.1 Effects of using Python and `numpy`

The runtimes seem quite large given the nature of the programs. Even for simple matrix multiplication, runtimes for $n > 512$ exceed a minute and a half.

One possible explanation is simply the nature of Python, which as a language typically runs more slowly than Java or C++. This could have contributed towards our larger runtimes across the board (i.e. not just for / not for matrices with powers of 2 dimensions). There is likely a more cache-conscious implementation in a different language in which the allocation and deallocation of memory is more explicity and more tightly-aligned with the current input size.

Additionally, my usage of `numpy` introduced a level of variability to my program: Python's many "under the hood" optimizations may have created inconsistencies between outputs with identical inputs.

6.1.2 Lower crossover point

The crossover value I identified for n_0, ≈ 11, was slightly lower than our predicted crossover point of 15. Due to the minimal/straightforward implementation of the standard multiplication algorithm, we turn to Strassen's to examine why our crossover point is lower than expected. One likely possibility is that using `numpy.add()` and `numpy.subtract()` greatly employs Python's "under the hood" optimizations. `Numpy`'s addition and subtraction functions are verifiably faster than those for lists: had I opted to use lists as my main data structure, my Strassen's implementation would almost certainly have been less efficient.) As a result, we would expect the Strassen's part of our algorithm to take less time than we analytically predict. This shorter run time would be conducive to switching over to standard matrix multiplication later: in other words, to a lower n_0.

6.1.3 Machine inconsistencies

I ran my code entirely on my local machine, which put great strain on my low-storage MacBook Pro. When I first started running code, I realized that the more tasks I performed while the code ran, the higher the resultant runtime, sometimes by margins as large as 10s. I ended up quitting most apps/operations besides my program (and an internet browser) in order to create consistency across tests.

6.1.4 Debugging difficulties

I ended up having to write multiple helper scripts and heavily employ the python3 terminal compiler to debug my functions and output errors. A particular point of difficulty was in attempting to properly implement "slicing" (i.e. submatrix index selection) for odd and padded matrices