Homework Batch I: Matrix Multiplication

Liberatori Benedetta

March 2021

Introduction

The file `matrix.py` includes the Python matrix template class and the Python implementations of the algorithms that will be explained in the following sections.

1

The function `strassen_matrix_mult` implements the Strassen's algorithm for the multiplication of 2^n x 2^n matrices. The algorithm relies on the Divide et Impera paradigm: partitions the matrices into 4 equally sized sub-matrices and goes on recursively until the sub-matrices become numbers. Yet it improves the naive Divide et Impera algorithm for this problem, which performs 8 recursive calls and has an asymptotic complexity of $\Theta(n^3)$, defining new matrices via additions and in the end it needs only 7 recursive calls. Therefore the asymptotic complexity is reduced to $\Theta(n^{log_2 7})$, at the expense of a non-constant amount of additional memory. The implementation provided switches to Gauss' standard method of matrix multiplication for small enough sub-matrices. This is motivated by the fact that Strassen's algorithm adds a considerable workload in additions and below a certain size it is less efficient than the standard one in terms of time-to-solution. Although the particular crossover size depends on implementation and hardware, it has been set to 32 since it is commonly considered a good choice. This do not change the asymptotic computational complexity because it just reduces the height of the recursion tree of the complexity of a factor which is constant.

2

The function `strassen_any_dim` implements the generalization of the Strassen's algorithm to any kind of matrix pair A^{nxm} and B^{mxp} using a padding technique. If the matrix in input are non-square or with dimensions which are not a power of 2 (or is pathological in both ways), then new square-matrices, with the smallest possible power-of-2 dimensions, containing the same elements

and additional rows/columns of zeros are defined. Then, the function calls `strassen_matrix_mult` with the new matrices.
It also takes into account that it could not be the case that both matrices need the padding, for example if the inputs are A^{nxn} and B^{nxp} with $n > p$ and n a power of 2. Then padding A would be a costly and unnecessary operation. In this case the multiplication using `strassen_matrix_mult` would be between the original A and the new matrix containing the elements of B.
This generalization enlarges the dimensions of the inputs and can require considerable more computation and memory allocation. Yet, the asymptotic time computational complexity remains the same. Firstly let us compute it for the non-power-of-2 square matrices case.
Let n be the dimension in input. Then two new square-matrices with dimension $m = 2^{\lceil log_2 n \rceil}$ are allocated. Now that the Strassen's algorithm can be used, the asymptotic computational complexity is $T(m) = \Theta(m^{log_2 7})$.
From the property of the ceiling function that $log_2 n \leq \lceil log_2 n \rceil < log_2 n + 1$, follows that $n \leq m < 2^{log_2 n+1} = 2n$. Then:

$$m^{log_2 7} < (2n)^{log_2 7} = 7n^{log_2 7} \Rightarrow m^{log_2 7} \in O(n^{log_2 7})$$

(1)

$$m^{log_2 7} \geq n^{log_2 7} \Rightarrow m^{log_2 7} \in \Omega(n^{log_2 7}) \quad (2)$$

And (1) and (2) give that $T(m) = \Theta(m^{log_2 7}) = \Theta(n^{log_2 7})$. This can be generalized to the case of non-square input matrices, computing the cost as a function of the biggest of the, at most tree, input dimensions. The computation in this new dimension is the same as above.

3

The function `strassen_less_memory` is an implementation of the Strassen's algorithm in which less additional memory is allocated. At each step in the recursion the number of auxiliary matrices is reduced to a minimum of a total of 8 sub-quadrants of A and B plus 1 matrix storing the results of the recursive calls. The 10 matrices used in `strassen_matrix_mult` to store the results of the sums are no more allocated and the sums are directly performed inside the recursive calls. The same happens for the 4 matrices storing the partial results of the multiplications. These partial results are directly computed inside the function that stores them in the final matrix. Finally, only one matrix is used to store the results of the recursive calls: each time the result is stored in the final matrix and then that auxiliary matrix can be reused. Compared to the original one, which used 29 additional matrices of halved dimensions at each step, requires just 9 of those. The number of recursive calls at each step remains 7 and also the cost of the operations needed at each step n remains in the order of $\Theta(n^2)$.

The two Python functions have been compared on square matrices of integers with dimensions power of 2 up to 2^{10}, for which the gain in execution time is near to 10%. The results obtained are here reported in seconds (using Python 3.8.5 64-bit). The code for the benchmark is reported as well in the `matrix.py`.

dim	strassen_matrix_mult	strassen_less_memory
1	0.000	0.000
2	0.000	0.000
4	0.000	0.000
8	0.001	0.001
16	0.004	0.004
32	0.023	0.013
64	0.086	0.077
128	0.626	0.458
256	3.434	3.302
512	25.691	23.615
1024	198.253	178.722

4

Let us measure the minimum amount of space needed for the Strassen's algorithm. In order for this to be minimum, the number of auxiliary matrices used in every recursive step must be reduced as much as possible. In addition to what already discussed in the previous section, all the auxiliary matrices storing the sub-quadrants of the input ones can be removed and queried directly from the original matrices. Once again the asymptotic cost complexity does not change; from the execution time point of view it did not improved the previous version (due to the additional calls of the function to obtain the sub-quadrants), thus the Python implementation of it had not been included in the code provided.
With this further reduction, the number of space allocated at each step is the one required for the two input matrices, the matrix storing the result and the one storing the result of the recursive calls (the same one for all the 7). Then, let S be a function of n of the space required, it follows that :

$$S(n) = \begin{cases} \Theta(1) & n = 1 \\ 3n^2 + S(\frac{n}{2}) & otherwise \end{cases}$$

The sum continues until $n/2^i = 1$, i.e. $i = log_2 n$ and thus the result is

$$S(n) = 3n^2 \sum_{i=0}^{log_2 n} \frac{1}{4^i} = 4n^2 \left(1 - (\frac{1}{4})^{log_2 n + 1}\right) = 4n^2 - 1 \in \Theta(n^2).$$

This can then be generalize to matrices of any dimensions, like those in section 2. For instance, the multiplication of square matrices of integers of dimension

1024 would require a number of bytes which is roughly 2^{22} multiplied by the number of bytes needed for integers, which is 4 in C/C++ and at least 24 on a 32-bit/64-bit system in Python.