Homework Batch I: Matrix Multiplication

Algorithmic Design - A.A. 2020-2021

Valentina Blasone

Due date: 25/03/2021

Instructions

Download the Python matrix template class from:

https://moodle2.units.it/mod/resource/view.php?id=213844

and:

1. implement the `strassen_matrix_mult` function to multiply two 2^n x 2^n matrices by using the Strassen's algorithm;
2. generalize `strassen_matrix_mult` to deal with any kind of matrix pair that can be multiplied (possibly also non-square matrices) and prove that the asymptotic complexity does not change;
3. improve the implementation of the function by reducing the number of auxiliary matrices and test the effects on the execution time;
4. answer to the following question: how much is the minimum auxiliary space required to evaluate the Strassen's algorithm? Motivate the answer.

Solution

1. The `strassen_matrix_mult` function is implemented in the script hw01_vb.py, provided together with this pdf file. The implementation reflects the one developed together in class. Some comments are included in the script.

2. The strategy followed to generalize the Strassen's algorithm to deal with any kind of matrix pair that can be multiplied, not necessarily squared, is as follows:

- The first observation is that the algorithm is directly applicable also to pairs of rectangular matrices, provided that all the dimensions are even. In the following I will then consider two matrices, `A` (n x m) and `B` (m x p), where n, m and p are all even numbers. In fact, by dividing `A` into four quadrants we obtain four matrices of same size (n/2 x m/2) and similarly for `B` we obtain four matrices of size (m/2 x p/2). The sum operations $S_1, \ldots S_{10}$ in the Strassen's algorithm are always performed among quadrants of the same matrix (either `A` or `B`) and therefore everything works fine also for rectangular matrices. The matrix multiplications $P_1, \ldots, P_7$ are always performed between a matrix of size (n/2 x m/2) and a matrix of size (m/2 x p/2) and thus we always obtain pairs of matrices that can be multiplied also in the rectangular case. Finally, but not less important, the correct result is preserved also for rectangular matrices.
- The only requirement is therefore to have matrices of even sizes at each recursive call of the function. This can be done by using padding on the original matrices to obtain two square matrices of size 2^n, where 2^n is the next power of two with respect to the maximum among original matrices dimensions. However, there is a more efficient strategy that consists in dynamically modifying the matrices at each call of the recursive function. To do so, at the beginning of the function, a section of code is added so that, in the case of an odd dimension a row/column of zeros is added to the matrix (padding). The central code is then maintained unchanged and at the end of the recursive call another section is added, this time to only retrieve as result the matrix of interest (i.e., of size (n x p) as from matrix multiplication between the original `A` and `B` matrices).

The function is implemented in the hw01_vb.py script as `strassen_matrix_mult_gen`.

To compute the asymptotic complexity we can start from the result for the original Strassen's algorithm: $O(n^{log_2\, n})$. In this general case, we can consider k as the maximum among all sizes of the two matrices `A, B` plus 1, to be conservative in the case the latter is odd:

$$k = \max(m, n, p) + 1 \tag{1}$$

The complexity of the general algorithm is exactly the one of Strassen's algorithm in which k substitutes n. If we identify n with $\max(m, n, p)$ it directly follows that the complexity is unchanged:

$$O(k^{log_2\, k}) = O((n+1)^{log_2\, (n+1)}) = O(n^{log_2\, n}) \tag{2}$$

3. To reduce the number of auxiliary memory, the following strategies are used:

   - reduce the auxiliary matrices needed to perform the partial matrix multiplications to only one matrix
   - assign the partial results directly to the final matrix `C`

The idea is to add the partial results for the final matrix quadrants $\mathtt{C}_{ij}$ dynamically, as results from the partial multiplications are ready. To do this, a unique auxiliary matrix is needed, `Aux`. The matrix `Aux` hosts the results of the temporary products ($\mathtt{P}_1, ..., \mathtt{P}_7$) once at the time. The result is added to the `C` matrix as soon it is available, by using either the `assign_submatrix` method or an additional method called `add_submatix`, appositely implemented. The auxiliary matrix is then used to store the next multiplication result. More details about the implementation can be found in the hw01_vb.py script, in the `strassen_matrix_mult_gen_opt` function.

We don't expect the execution time to decrease with respect to the general implementation of the algorithm, given the fact that the number of recursive calls is unchanged. The comparison is done by usign the `timeit` function in python. Table 1 shows the results obtained in terms of time for increasing-size squared matrices, with power of 2 size. Table 2 shows instead the same comparison but for pairs of generic matrix that can be multiplied, considering first two squared matrices, which size is however not a power of 2 and finally the most general case of two rectangular matrices of different and odd sizes. As expected, we can see that the timings for the memory-optimized algorithm are very similar to the ones of the standard general Strassen algorithm, only slightly lower. In both cases the Strassen algorithm slightly outperforms the Gauss algorithm, with improvement more evident for higher matrices dimensions.

Table 1: Comparison of times for matrices of size 2^i for different values of i, using the different algorithms. Gauss is the Gauss algorithm, Strassen G is the general Strassen algorithm and Strassen G-M is the memory-optimized Strassen algorithm. All measures of time are in seconds.

`n`	Gauss	Strassen G	Strassen G-M
$2^1 = 2$	0.000	0.000	0.000
$2^2 = 4$	0.000	0.000	0.000
$2^3 = 8$	0.000	0.000	0.000
$2^4 = 16$	0.001	0.001	0.001
$2^5 = 32$	0.008	0.009	0.008
$2^6 = 64$	0.058	0.067	0.063
$2^7 = 128$	0.457	0.497	0.467
$2^8 = 256$	3.705	3.641	3.343
$2^9 = 512$	32.456	26.660	24.875

Table 2: Comparison of times for generic matrices of different sizes, using the different algorithms. Gauss is the Gauss algorithm, Strassen G is the general Strassen algorithm and Strassen G-M is the memory-optimized Strassen algorithm. All measures of time are in seconds.

$\mathtt{n}_A$	$\mathtt{m}_A$	$\mathtt{n}_B$	$\mathtt{m}_B$	Gauss	Strassen G	Strassen G-M
300	300	300	300	6.290	5.808	5.500
351	351	351	351	10.275	8.608	8.038
412	327	327	283	8.967	8.037	7.565

In the class `Matrix` there is a method called `compare_matrix`, created to compare the object to a given matrix, by calculating the difference between each term and checking that it is smaller than a given tolerance. The method is used to check all the functions implementations, with a tolerance of $1e-10$, taking as benchmark the Gauss algorithm result.

4. The minimum auxiliary space required to evaluate the Strassen's algorithm can be evaluated by considering the following:

- In each call of `strassen_matrix_mult_gen_opt` we need one auxiliary matrix of dimensions $\mathtt{n}_C/2$ x $\mathtt{m}_C/2$, where $\mathtt{n}_C$ and $\mathtt{m}_C$ are the number of rows and columns the matrix C, result of the function call. The auxiliary matrix is the one in which the partial products are stored successively. The total is $(n/2)^2$.
- If we call $M(n)$ the amount of memory required to evaluate two n x n matrices, the additional space required by the first recursion is therefore $M((n/2)^2)$.
- The total amount of memory considering all recursions is therefore (k is considered as in Equation (1)):

$$
\begin{aligned}
M(k) &= \left(\frac{k}{2}\right)^2 + M\left(\frac{k}{4}\right)^2 \\
&= \left(\frac{k}{2}\right)^2 + M\left(\frac{k}{4}\right)^2 + M\left(\frac{k}{8}\right)^2 \\
&= \sum_i \left(\frac{k}{2^i}\right)^2 = O(k^2)
\end{aligned}
$$