Homework Batch I: Matrix Multiplication

Solutions

April 17, 2021

Later on, M_{ij} will denote one of the M's quadrants. In particular,

$$M = \left[\begin{array}{c:c} M_{11} & M_{12} \\ \hdashline M_{21} & M_{22} \end{array}\right]$$

1. The implementation of the Strassen's algorithm was detailed during the 4th lesson of the course.

2. Let A and B two matrices having n rows and m columns and m rows and o columns, respectively. Thus, the matrix $C = A \times B$ has n rows and o columns.

   It is easy to prove that the matrix-based equations at the based of Strassen's algorithm hold for any A and B that can be split in quadrants, i.e., n, m, and o must be even.

   If one among n, m, and o is odd, we can use **padding** and add either a new row a new column to A or B still been able to compute C. As a matter of fact, if n is odd, then we can add one row to A and get:

   $$\left[\begin{array}{c} A \\ \hdashline 0 \quad \dots \quad 0 \end{array}\right] \times B = \left[\begin{array}{c} C \\ \hdashline 0 \quad \dots \quad 0 \end{array}\right], \tag{1}$$

   if o is odd, then we can add one column to B and get:

   $$A \times \left[\begin{array}{c:c} B & \begin{matrix} 0 \\ \vdots \\ 0 \end{matrix} \end{array}\right] = \left[\begin{array}{c:c} C & \begin{matrix} 0 \\ \vdots \\ 0 \end{matrix} \end{array}\right], \tag{2}$$

   and if m is odd, then we can add one column to A and one row to B and get the following equation:

   $$\left[\begin{array}{c:c} A & \begin{matrix} 0 \\ \vdots \\ 0 \end{matrix} \end{array}\right] \times \left[\begin{array}{c} B \\ \hdashline 0 \quad \dots \quad 0 \end{array}\right] = C. \tag{3}$$

Thus, in order to extend the Strassen's algorithm to non-square matrices, we only have to check at the beginning of each call that n, m, and o are even and, whenever this is not the case, to use padding to reduce the problem to a problem involving matrices satisfying above condition.

The recursive step can be applied as long as $\min(m, n, o) > 1$. Thus, we can choose $\min(m, n, o) = 1$ as base case condition.

Copying an $r \times c$ matrix into a new $(r+1) \times (c+1)$ matrix and the sums and subtractions between $r \times c$ matrices cost $\Theta(r * c)$. It follows that the recursive equation describing the complexity of this new version of the algorithm is:

$$T_S(n, m, o) = \begin{cases} \Theta(1) & \text{if } \min(n, m, o) = 1 \\ 7 * T_S\left(\left\lceil \frac{n}{2} \right\rceil, \left\lceil \frac{m}{2} \right\rceil, \left\lceil \frac{o}{2} \right\rceil\right) + \Theta(\gamma) & \text{if } \min(n, m, o) > 1 \end{cases}$$

where $\gamma = n * m + m * o + n * o$. By using the recursion tree, it is easy to prove that $T_S(n, m, o) \in \Theta(\gamma * \min(n, m, o)^{(\log_2 7)-2})$. Hence, if $\min(n, m, o) = o$, then $T_S(n, m, o) \in \Theta(n * m * o^{(\log_2 7)-2})$. It is worth to notice that the best improvement with respect to the Gauss's algorithm occurs when $\min(n, m, o) = \max(n, m, o)$ and $n = m = o$.

3. There are many ways of solving this exercise and the mark depends on how much efficient is your solution. Because of it, I will focus on the solution that requires the smallest amount of auxiliary space. However, there are much easier solutions that can get up to the 80% of the mark. Please, notice that the Exercise did not require a description of the solution as you will find in the following, but exclusively its Python implementation.

   By resorting the code of Exercise 2's solution, we can easily reduce the number of $(n/2) \times (n/2)$ auxiliary matrices to 15: the sub-matrices $A_{11} \ldots A_{22}$, $B_{11} \ldots B_{22}$, $C_{11} \ldots C_{22}$, and, at most, 3 matrices per recursive call (i.e., two actual parameters and the output). The matrices A_{ij}, B_{ij}, and C_{ij} are sub-matrix of the formal parameters and the output of the algorithm. So, the values in A_{ij} and B_{ij} are originally stored in A and B, respectively, while the numbers assigned to C_{ij} will eventually end up in C. Thus, if we find a way to refer to the cells of A_{ij} ($B_{i,j}$ and $C_{i,j}$) directly in A (B and C, respectively), we can avoid to allocate the space for the sub-matrices A_{ij} ($B_{i,j}$ and $C_{i,j}$, respectively).

   Let us assume that both n and m are even and, for the sake of example, let us focus on the sub-matrix A_{21}. The value contained in the i-th row and j-th column of A_{21} (i.e., $A_{21}[i, j]$) was stored in A in the $(i + n/2)$-th row and j-th column. Thus, in order to mimic A_{21}'s behaviour and access its values, we only need a reference to A, both the row index and the column index in A that correspond to A_{21}'s first row and column, respectively, (i.e., $n/2 + 1$ and 1) and, for completeness, the number of A_{21}'s rows and columns (i.e., $n/2$ and $m/2$). Analogous considerations hold for A_{ij}, B_{ij}, and C_{ij}. By using this idea, we can avoid to allocate the space required

by A_{ij}, B_{ij}, and C_{ij}. Since the technique described above is just like focusing on a part of a matrix and seeing it as a sub-matrix of the former even tough the sub-matrix is not really allocated, the data structure that implements it may be called `MatrixView`.

Due to the padding, the solution of Exercise 2 may require further three auxiliary matrices: one for the padding of A, one for that of B, and one as result of the multiplication between the padded A and the padded B.

Notably, the very same idea that has been used to avoid the allocation of the sub-matrices can also be used to avoid the padding. Let us assume n to be odd. If $i \in [1, \lceil n/2 \rceil - 1]$, then all above considerations about A_{21} and its values holds too and the value $A_{21}[i, j]$ is equal to the value $A[i + \lceil n/2 \rceil, j]$. As far as the case $i = \lceil n/2 \rceil$ may concern, the $(\lceil n/2 \rceil + \lceil n/2 \rceil)$-th row does not exist in A when n is odd and, so, any reference to the $\lceil n/2 \rceil$-th row in A_{21} would end up with an error. However, if, in addition to `MatrixView` members, we also maintain the number of rows and columns in A_{21} that are not due to padding, then we can "emulate" the behaviour of a padded matrix: whenever we are trying to read a value from a row and a column that is not due to padding, we will access to the corresponding value in A, in the other cases, the value will be 0.

In conclusion, no matter how large is the sub-matrix we need to represent, `MatrixView` will always maintain seven values:

`matrix` a reference to the data structure that actually maintains in memory the matrix y the matrix;

`first_row` **and** `first_col` the the indexes on `matrix` of the `MatrixView`'s first row and column;

`num_of_rows` **and** `num_of_cols` the number of accessible rows and columns on `MatrixView`;

`real_rows` **and** `real_cols` the number of `MatrixView`'s rows and columns that are not due to padding.

See the link at the end of this document to download the code.

4. The implementation proposed as solution of Exercise 3 uses at most three auxiliary matrices per recursive call: 2 devoted to store the call actual parameters and 1 for the result. Since each recursive call involves matrices whose dimensions are half of the original ones, the overall auxiliary space required to compute the Strassen's algorithm is:

$$\begin{aligned} S_S(n, m, o) = & S_S\left(\left\lceil \frac{n}{2} \right\rceil, \left\lceil \frac{m}{2} \right\rceil, \left\lceil \frac{o}{2} \right\rceil\right) + \\ & + \left\lceil \frac{n}{2} \right\rceil * \left\lceil \frac{m}{2} \right\rceil + \left\lceil \frac{m}{2} \right\rceil * \left\lceil \frac{o}{2} \right\rceil + \left\lceil \frac{n}{2} \right\rceil * \left\lceil \frac{o}{2} \right\rceil + \\ & + \Theta(1) \end{aligned}$$

where n, m, and o are the numbers of A's rows, A's columns, and B's columns, respectively. By using the substitution method, it is easy to prove that $S_S(n, m, o) \in O(n * m + m * o + n * o)$.

A brutal implementation of all required code can be downloaded here.