Advanced Algorithm Project : Matrix Maximum Segment Problem
Report

Louis NICOLAS
Charles FRANCHI
Ugo LABBÉ
Alejandro CARVAJAL MONTEALEGRE
Mohadeseh(Fatemeh) GOL POUR

December 3, 2023

Abstract

This report presents a comprehensive study of the two-dimensional maximum matrix segment problem, which aims to find a sub-matrix with the largest sum, given a matrix as input. The problem is approached in two different scenarios: an unconstrained version and a constrained version. The distinction between these methods lies in the fact that the constrained version takes the size of the sub-matrix with the largest sum as an additional input.

The problem was addressed using various algorithms, including brute-force, branch and bound, greedy, dynamic programming, randomized, ant colony, and divide-and-conquer approach. Each algorithm explores the solution differently, resulting in distinct time complexities. Additionally, in some algorithms, the output sub-matrix may vary, as certain approaches do not necessarily yield the best solution but rather a local optimal one. This project evaluates all the aforementioned methods, comparing their performance in terms of algorithm execution time, complexity, and output.

Contents

1 Introduction

The *Matrix Maximum Segment Problem* (MMSP) is a well-known problem generalization of the *Maximum Sub-array Problem* which is easily solvable. The goal is to find the optimal sub-matrix in a matrix such the sum of all elements of the sub-matrix is maximum. For example, if you take the matrix :

$$A = \begin{pmatrix} -4 & -5 & -6 & 2 \\ 2 & 10 & 5 & -2 \\ -3 & 4 & -1 & 1 \end{pmatrix}$$

The optimal sub-matrix is :

$$\begin{pmatrix} 10 & 5 \\ 4 & -1 \end{pmatrix}$$

and the sum is 18.

It's important to notice a matrix can have several solution to the MMSP, like in this case :

$$B = \begin{pmatrix} 1 & 2 \\ 2 & -2 \end{pmatrix},\ Sol1 = \begin{pmatrix} 1 & 2 \end{pmatrix},\ Sol2 = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, Sol3 = \begin{pmatrix} 1 & 2 \\ 2 & -2 \end{pmatrix}$$

To this problem, we can add a constrained on the size of the sub-matrix, to be a specific $(K \times L)$ one. For example, with the matrix A, if we add the constrained the sub-matrix is (1×3), then the solution is :

$$Sol(1,3) = \begin{pmatrix} 2 & 10 & 5 \end{pmatrix}$$

More formally, the MMSP problem can be define as follow : Let $M \in \mathcal{M}_{m,n}(\Re)$ a matrix.

$$MMSP(M) = \max_{1 \leq l1 \leq l2 \leq m,\ 1 \leq k1 \leq k2 \leq n} \sum_{i=l1}^{l2} \sum_{j=k1}^{k2} M_{i,j}$$

Let $K \in [\![1, m]\!]$ and $L \in [\![1, n]\!]$. The constrained version is :

$$MMSP(M, K, L) = \max_{1 \leq r \leq m-K+1,\ 1 \leq c \leq n-L+1} \sum_{i=r}^{r+K-1} \sum_{j=c}^{c+L-1} M_{i,j}$$

The goal of this project was to design several algorithms solving the unconstrained version, and other algorithms solving the constrained version. In both case, we focus in finding only one maximum subarray if several exists.

2 Theoretical Framework

In this section, the principle of the methods used in this report will be explained. We then explain how we use these methods to solve the MMSP, for both unconstrained and constrained versions, as well as its functioning and possible output.

2.1 Brute-force

Brute-force systematically explores all possible solutions without employing heuristics or sophisticated strategies. In the pursuit of identifying the sub-matrix with the maximal sum within a given input matrix, the algorithm iterates through every possible combination of starting and ending indices for both rows and columns, calculating the sum of each sub-matrix encountered. By exhaustively evaluating every conceivable configuration, the algorithm ensures a comprehensive search of the solution space and for this reason, the method tends to have a higher time complexity compared to more refined algorithms. Despite this drawback, its deterministic nature guarantees an optimal solution.

2.1.1 Unconstrained version

As explained above, we simply compute every submatrices.

Pseudo-code

```
bruteforce(matrix):
Begin
    Let m,n be the dimensions of the matrix
    Let indexes be the 4 indexes of the maximum sub-matrix.
    max <- - inf

    For l1 <- 1 to m - 1
        For c1 <- 1 to n - 1
            For l2  <- l1 to m
                For c2  <- c1 to n

                    newsum  <- the sum of all
                    \ coefficients of the submatrix from (l1,c1) to (l2, c2)

                    If newsum > max
                        max  <- newsum
                        indexes  <- (l1, l2, c1, c2)
    Return max, indexes
End
```

Complexity Let $M \in \mathcal{M}_{m,n}(\Re)$ a matrix. The number of operations for **bruteforce** is :

$$
\begin{aligned}
T(m,n) &= \Theta(1) + \sum_{l1=1}^{m}\sum_{c1=1}^{n}\sum_{l2=l1}^{m}\sum_{c2=c1}^{n}[(c2-c1)\times(l2-l1)+\Theta(1)] \\
&= \Theta(m^2n^2) + \sum_{l1=1}^{m}\sum_{l2=l1}^{m}(l2-l1)\times\sum_{c1=1}^{n} -c1\times(n-c1+1) + \left(\frac{n(n+1)}{2} - \frac{(c1-1)c1}{2}\right) \\
&= \Theta(m^2n^2) + \sum_{l1=1}^{m}\sum_{l2=l1}^{m}(l2-l1)\times\sum_{c1=1}^{n}\frac{c1^2}{2} - c1\left(n+\frac{1}{2}\right) + \frac{n(n+1)}{2} \\
&= \Theta(m^2n^2) + \sum_{l1=1}^{m}\sum_{l2=l1}^{m}(l2-l1)\times\left(\frac{n(n+1)(2n+2)}{12} - \frac{n(n+1)}{2}\left(n+\frac{1}{2}\right) + n\frac{n(n+1)}{2}\right) \\
&= \Theta(m^2n^2) + \sum_{l1=1}^{m}\sum_{l2=l1}^{m}(l2-l1)\times\left(\frac{n^3}{6} + \Theta(n^2)\right) \\
&= \Theta(m^2n^2) + \sum_{l1=1}^{m}\sum_{l2=l1}^{m}(l2-l1)\times\Theta(n^3) \\
&= \Theta(m^2n^2) + \Theta(n^3)\times\sum_{l1=1}^{m}\sum_{l2=l1}^{m}(l2-l1) \\
&= \Theta(m^2n^2) + \Theta(n^3)\Theta(m^3) \\
&= \Theta(m^3n^3)
\end{aligned}
$$

with $(c2-c1)\times(l2-l1)$ the number of sum to compute the sum of the submatrix.

2.1.2 Constrained version

Complexity Let $M \in \mathcal{M}_{m,n}(\Re)$ a matrix, K the number of rows and L the number columns of the output sub-matrix. The number of operations for the constrained version is :

$$\begin{aligned} T(m,n) &= \Theta(1) + \sum_{r=1}^{m-K+1} \sum_{c=1}^{n-L+1} KL \\ &= \Theta(1) + KL \times (m-K+1) \times (n-L+1) \\ &= \Theta(1) + KL(mn - mL + m - nK + KL - K + n - L + 1) \\ T(m,n) &= \Theta(KL(mn + KL)) \end{aligned}$$

So finally, $T(m,n) = \Omega(mn)$ when $K = L = 1$, and $T(m,n) = O(m^2n^2)$ when $K = m$ and $L = n$.

2.2 Branch-and-Bound

The Branch-and-Bound approach explores the solution space through a process known as branching. The algorithm divides the problem into smaller, more manageable sub-problems or branches. Subsequently, using a criterion introduced by the algorithm, the least promising branches, which are unlikely to lead to a solution, are pruned. For the problem of identifying a sub-matrix with the maximal sum given an input matrix,

2.2.1 Applications of this Approach

- **Unconstrained Version**

  The algorithm implemented for solving the MMSP begins by setting up initial parameters, such as the dimensions of the input matrix, the initial values for the maximum sum and maximum subarray, as well as the position of the submatrix. It then defines a function to calculate the sum of a subarray, checks if a subarray is within the matrix bounds, and generates child nodes for a given node. An iteration is made, considered the main loop of the code, to find all possible starting points for the subarray. It creates a priority queue where nodes are dequeued based on the sum of the subarray, prioritizing nodes with a higher potential for a larger sum. After exploring the queue, the function returns the subarray with the largest sum and its coordinates.

- **Constrained Version**

  The main difference with the unconstrained version is that the value of the submatrix $K * L$ is given as an input, and the algorithm has to find a submatrix of that size with the largest sum. The algorithm implemented for this section first initializes and stores the values of the maximum sum, submatrix, and indices. Then, it calculates the sum of the submatrix and stores the indices. A function with the goal of bounding checks if the current branch can lead to a better solution, with criteria that the dimension is within bounds and if the current sum, plus the potential maximum sum, is greater than the current result's maximum sum. The next step is a function that explores different branches of the search and prunes branches that are not promising based on the criteria of the previous step. An iteration is then done over all possible starting positions for the submatrix with dimensions $K * L$. Finally, the submatrix, indices, and sum value are printed.

2.2.2 Complexity

- **Unconstrained version**

  The time complexity for the uncontrained version is:

$$T(m,n) = O(M \cdot N \cdot \log(M \cdot N))$$

where:

- M : number of rows of the input matrix A.
- N : number of columns of the input matrix A.

The nested loops add a time complexity of $O(M \cdot N)$. The outer loop runs over all possible starting points for the subarray and the inner loop processes the priority queue. The process of enqueueing and dequeuing in a priority queue has a time complexity of $O(logK)$, where K is the number of elements in the queue. In worst case scenario, the number of nodes is proportional to the size of the matrix, making this term $O(Log(M \cdot N))$.

The worst case scenario would be when the input matrix has a structure where each element is significantly different from its neighbors, ensuring that the algorithm needs to explore many submatrices to find the one with the maximum sum

- **Constrained version**

  The time complexity of the contrained version is:

$$T(m,n) = O((M-K+1)(N-L+1) \cdot KL + M \cdot N \cdot 2^{(M+N)})$$

  where:

  - M : number of rows of the input matrix A.
  - N : number of columns of the input matrix A.
  - K : number of rows in the submatrix for which we are calculating the maximum sum.
  - L : number of columns in the submatrix for which we are calculating the maximum sum.

  The reason for this time complexity is the nested loops. The outer loop iterates $M - K + 1$ times and the inner loop iterates $N - K + 1$ times. Inside the nested loops, a function that calculates the sum is called, which has a time complexity of $O(K \cdot L)$. The function that explores all possible branches in the search space. In worst case, at each position, it cans continue branching to the right or downward. This adds an exponential time complexity $2^{(M+N)}$. The consult to the array that contains the submatrix sums has an overall time complexity $O(M \cdot N)$ for the whole execution as it may perfomr $M \cdot N$ times.

  In $Big - O$ notation, the impact of the term $O(M-K+1)(N-L+1) \cdot KL$ in the time complexity is irrelevant, compared to the exponential factor present the overall time complexity. For that reason, the time complexity can be simplified and expressed as:

$$O(M \cdot N \cdot 2^{M+N}).$$

  The worst case scenario would be when the algorithm doesn't find pruning oportunities and must check all possible combinations of submatrices. This is more likely to occur when the matrix has a large number of rows and columns, and the submatrix $K * L$ is small.

2.3 Greedy Algorithms

A greedy algorithm is a strategic problem-solving methodology that operates by making locally optimal choices at each stage, aiming to achieve an overall optimal solution if possible. It focuses on selecting the best available option in the immediate context without considering potential long-term implications. In the context of identifying the sub-matrix with the largest sum within an input matrix, the greedy approach iteratively selects the sub-matrix with the maximum sum at each step, disregarding potential future consequences. It's essential to note that decision-making process may not always lead to the globally optimal solution.

2.3.1 Application of this approach on the MMSP

- **Unconstrained version**

  The principle of greedy approach for the MMSP is selecting the best submatrix at the time. First we select the best 1,1-submatrix ergo the element with the highest value. We will then check the sum of every possible matrix with an increased height or width of 1, which means we will have 4 candidates : one matrix with a row above, one matrix with a row below, one matrix with a column on the left and one matrix with a column on the right. The matrix with the highest sum will become our new submatrix and we will iterate over it. The iteration stops when no candidates has an highest sum than the current submatrix.

- **Constrained version**

  The constrained version is somewhat similar to the unconstrained version but we will not include new candidates if they do not match the constraint which means we will not include new row to the candidates if the maximum height is fulfilled and we will not include new columns if the maximum width is fulfilled. The end of the iteration is also different, this time we do not check if the sum decreases, we instead check if all the requirements are fulfilled.

2.3.2 Complexity

- **Unconstrained version**

  Outside of the iterative part, we have a sum of complexity O(mn). In every iteration, we have to calculate the sum of the row above, the row below, the column in the right and the column in the left, so in the worst case we have a calculation of $O(m+n)$ for a matrix $\mathcal{M}_{m,n}$, we do this calculation a maximum of $m \times n$ which gives us a total of :

$$\begin{aligned} T(m,n) &= mn(2\sum_{r=1}^{m} O(r) + 2\sum_{c=1}^{n} O(c) + O(1)) + O(mn) + O(1) \\ &= mn(O(2(m+n)) + O(1)) + O(mn) + O(1) \\ &= O(2mn(m+n)) + 2O(mn) + O(1) \\ &= O(mn(m+n)) \end{aligned}$$

  The best case is if we don't find a better submatrix than the maximum value of the matrix at the first iteration which makes the iteration part $\Omega(1)$ so we have a complexity of :

$$\begin{aligned} T(m,n) &= \Omega(mn) + \Omega(1) \\ &= \Omega(mn) \end{aligned}$$

- **Constrained version**

  The constrained version works the same as the normal version but it stops when the matrix is the good size so we have an exact time complexity. For a submatrix $\mathcal{M}_{K\times L}$ where K is the height of the matrix and L is the length of the matrix, we have a complexity of :

$$\begin{aligned} T(m,n) &= KL(2\sum_{r=1}^{K} \Theta(r) + 2\sum_{c=1}^{L} \Theta(c) + \Theta(1)) + \Theta(mn) + \Theta(1) \\ &= KL(\Theta(2(K+L)) + \Theta(1)) + \Theta(mn) + \Theta(1) \\ &= \Theta(2KL(K+L)) + \Theta(KL) + \Theta(mn) + \Theta(1) \\ &= \Theta(mn(K+L)) \end{aligned}$$

  Thus, we obtain the best scenario if $K = L = 1$, so $T(m,n) = \Omega(mn)$ and the worst scenario if $K = m$ and $L = n$, so $T(m,n) = O(mn(m+n))$.

2.3.3 Criticism on this approach

While this approach is fast with a complexity of $O(mn(m+n))$, it does not give the optimal solution each time. It is especially not efficient with big matrices where it can often give a solution that is far from the optimal one.

2.4 Divide and conquer

A divide and conquer algorithm is a recursive method such we divide an original problem in small subproblems recursively, until we found basic cases easily solvable, and then construct the global solution by assembling all the subproblems together. For the MMSP, we divide the matrix in two among the two dimensions until we have only 1 element matrix, and use a trick for submatrix straddling on two parts.

2.4.1 Principle

The main idea is to recursively divide the matrix in two parts, so the solution is compulsorily in the first part, the second one, or precisely in both.

1. For the two seperate parts, we divide it recursively, with first a left to right split, then a up to bottom split (on 1D array so).

2. For the part on both side, we do a recursive approach, redevided the matrix in the opposite oritentation (up-bottom if it was left-right) and search the max submatrices in the first part, the second one, and both (in a similar way than the general function, but we have now a middle point and not a middle line).

We save calculus by going lineary through the submatrix with middle constraint : for a 1D-array, the maximum subarray passing through a middle element is the sum of the maximum subarray ending with the middle element and the maximum subarray starting with the middle + 1 element.
Indeed, if A is a 1D-array of size n, and S the maximum subarray passing through the mid^{th} element. So

$$\exists j, k \leq n,\ j \leq mid,\ k \geq mid+1,\ S = \sum_{i=j}^{k} A_i$$

Thus, $\sum_{i=j}^{mid} A_i$ is the maximum subarray ending with the mid^{th} element. Else, it would exist $j' \neq j$ such that the sum of the element from j' to mid would be greater, and by adding the element of A from $mid+1$ to k to this sum, we would find a greater subarray than S, so it would not be maximum.
We can apply the same logic for $\sum_{i=mid+1}^{k} A_i$.
Thus, by construction, S is exactly the sum of this two maximum subarrays.
These maximums can be find in linear time, adding the next element to a stored sum of the previous one.

Such we can use the *sumrc* technic (presented in the Dynamic part 2.5.2) to have a global $\Theta(mn)$ complexity and then a $\Theta(1)$ complexity for each subarray sum, storing the element of the previous sum is not really usefull, but we still save some computation by finding the maximum with middle line in linear time.

For 2D-arrays with a center element, we apply this logic with each combination of columns on either sides of the middle element, transforming it in a 1D-array by summing element of each lines, and finally extract the maximum among those $m \times (m+1)/2$ subarrays.

2.4.2 Complexity

Let M be the matrix of size $(m \times n)$. Then the next algorithm is going through 2 recursives algorithms :

$$
\begin{aligned}
T(m,n) &= 2T(m/2,n) + Midline(m,n) + \Theta(1) \\
T(1,n) &= 2T(1,n/2) + Midcolumn(n) + \Theta(1) \\
&= 2T(1,n/2) + maxarray(n) + \Theta(1) \\
&= 2T(1,n/2) + \Theta(n) \\
T(1,1) &= \Theta(1)
\end{aligned}
$$

$$
\begin{aligned}
Midline(m,n) &= 2Midline(m,n/2) + Midmid(m,n) + \Theta(1) \\
&= 2Midline(m,n/2) + \Theta(mn^2) \\
Midline(m,1) &= maxarray(m) = \Theta(m)
\end{aligned}
$$

because $maxarray$ is linear as explained above, and $Midmid$ is the test of all pair of columns around the middle element, so it's $(m/2)^2 \times n$ because we use $maxarray$ to find the max. Thus, with the Master Theorem complex formulation,

$$Midline(m,n) = \Theta(mn^2)$$

because m is the same in all the recurrence, so it can be factorize in Θ, and $\Theta(n^2) = \Omega(n^{\log_2 2+1})$ and $2\Theta(n^2)/2 = \Theta(n^2)$ (third property).

$$T(1,n) = \Theta(n \log n)$$

because $\Theta(n) = \Theta(n^{\log_2 2} \log^0 n)$ (second property).

So we finally get :

$$
\begin{aligned}
T(m,n) &= 2T(m/2,n) + \Theta(mn^2) \\
T(1,n) &= \Theta(n \log n) \\
\Leftrightarrow T(m,n) &= \Theta(m \log(m) n^2)
\end{aligned}
$$

because n^2 can be factorize, $\Theta(m) = \Theta(m^{\log_2 2} \log^0 m)$ (second property), and the initial $\Theta(n \log n))$ is absorbed by the global complexity ($\Theta(n \log n) = O(n^2)$).

2.5 Dynamic Programming : first approach

Dynamic programming is a technique that involves breaking down a complex problem into smaller overlapping sub-problems and solving each sub-problem only once, storing the solutions to avoid redundant computations.

2.5.1 Theoric principle for a totally dynamic algorithm

Let $M \in \mathcal{M}_{m,n}(\Re)$ a matrix.
Let $S[i,j]$ be the sum of the maximum submatrix among submatrices with maximum size (i,j) in M.
Let $exactS[i,j]$ be the sum of the maximum submatrix among submatrices of exact size (i,j) in M (thus, $exactS$ is just the constrained version).
Then :

$$
\begin{aligned}
\forall (i,j) \in [\![2,m]\!] \times [\![2,n]\!] : \quad & S[i,j] = max(S[i-1,j],\ S[i,j-1],\ exactS[i,j]) \\
\forall i \in [\![2,m]\!] : \quad & S[i,1] = max(S[i-1,1],\ exactS[i,1]) \\
\forall j \in [\![2,n]\!] : \quad & S[1,j] = max(S[1,j-1],\ exactS[1,j]) \\
& S[1,1] = exactS[1,1] = \max_{i,j} M[i,j]
\end{aligned}
$$

define an optimal subproblem.

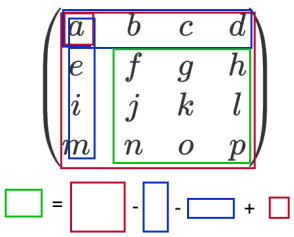
Proof of optimality If S is the optimal sub matrix, let (i_{opt}, j_{opt}) is size, then $S = exactS[i_{opt}, j_{opt}]$ and $\forall i, j: \quad S \geq S[i,j], S \geq exactS[i,j]$ by definition, so it will be selected or an equivalent one with exactly the same sum.

2.5.2 *exactS* : Constrained version

to compute $exactS$, we create a cumulate of the sum coeffcient form the top left corner $(0,0)$ to the bottom right corner (m,n).

For example,

$$mat = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}, sumrc = \begin{pmatrix} a & a+b & a+b+c \\ a+d & a+b+d+e & a+b+c+d+e+f \\ a+d+g & a+b+d+e+g+h & a+b+c+d+e+f+g+h+i \end{pmatrix}$$



We can define it in that way : $sumrc$ is defined for indices $(i,j) \in [\![-1;m]\!] \times [\![-1;n]\!]$. $\forall (i,j) \in [\![0;m]\!] \times [\![0;n]\!]$, $sumrc[i,j]$ is the sum of all coeff from $[0,0]$ to $[i,j]$.
If i or $j = -1$, $sumrc[i,j] = 0$.

Then the sum of all elements from (i,j) to (k,l) is $sumrc[k,l] - sumrc[k-i-1,l] - sumrc[k,l-j-1] + sumrc[k-i-1,l-j-1]$, which is $\Theta(1)$ for each submatrix.

Compute this matrix is linear, so it's $\Theta(mn)$. This method cans be seen as dynamic such we store some sum to not recompute it all the times.

Complexity With this matrix, you just need to compute all submatrices of exact size (K,L), so it's maximum $m \times n$ subamtrices, so the complexity is $\Theta(mn)$ (because in every cases we need to compute the $sumrc$ matrix).

Recurence beetween different *exactS* ? $exact[i,j]$ can't be construct from $exact[i-1,j]$ cause it cans be surounded by big negative elements.

Counter-example :

$$\begin{pmatrix} -100 & 4 & 4 \\ 20 & 4 & 4 \\ 20 & 4 & 4 \\ -100 & 4 & 4 \end{pmatrix}$$

$exactS[2,1] = 40$ but unfortunetely $exactS[3,1] = 12$.

2.5.3 Pseudo-code

```
dynamic(matrix)
Begin
    Compute sumrc.
    Let S, exactS and exactSind be as define before this section.
    Let indices be a matrix of indices of size (m * n)
    indices[1,1] <- exactSind[1,1]

    For i <- 2 to m
        If S[i-1,1] > exactS[i,1]
            S[i,1] <- S[i-1,1]
            indices[i,1] <- indices[i-1,1]
        Else
            S[i,1] <- exactS[i,1]
            indices[i,1] <- exactSind[i,1]
```

```
    For j <- 2 to n
        If S[1,j-1]  exactS[1,j]
            S[1,j] <- S[1,j-1]
            indices[1,j] <- indices[1,j-1]
        Else
            S[1,j] <- exactS[1,j]
            indices[1,j] <- exactSind[1,j]

    For i <- 2 to m
        For j <- 2 to n
            S[i,j] <- \max(S[i,j-1], S[i-1,j], exactS[i,j])
            indices[i,j] is respictively indices[i,j-1], indices[i-1,j], exactSind[i,j]

    Return S[m,n], indices
End
```

Complexity Such compute *smrc* or *exactS* is $\Theta(mn)$,

$$\begin{aligned} T(n,m) &= \Theta(mn) + \sum_{i=2}^{m} \Theta(mn) + \sum_{j=2}^{n} \Theta(mn) + \sum_{i=2}^{m}\sum_{j=2}^{n} \Theta(mn) \\ &= \Theta(m^2n^2) \end{aligned}$$

2.5.4 Criticism on the totally dynamic approach

Except the computation of *exactS*, the way this algorithm is dynamic is only on how we compare and find the maximum. In fact, we don't save any computation, so use dynamic here is a bit overkill.

Such there seems to not have any subproblems rules such the algorithm is totally dynamic from begin to end, we will adapt the Kadane's algorithm to use a dynamic approach in a more brute algorithm to save time complexity.

2.6 Dynamic Programming : 2D-Kadane approach

Kadane's algorithm is a linear algorithm for 1D array, for finding the max sub-array sum (source). The idea is to use the *exactS* method to compute a sum matrices, and then to create 1D-array with columns sums that can be solved with Kadane's algorithm.

2.6.1 Dynamic principle

1. Characterize the structure of an optimal solution : the optimal solution is the $\max_{1\leq i\leq n} maxending[i]$, with $maxending[i]$ the maximum sub-array ending with the index i.

2. Divide in subproblem :

$$\begin{aligned} \forall i \in [\![2,n]\!],\ maxending[i] &= \max(maxending[i-1] + A[i], A[i]) \\ maxending[1] &= A[1] \end{aligned}$$

   Proof by contradiction : let suppose $\sum_{k=j}^{i} A[k] = \max(maxending[i-1] + A[i], A[i])$ is not the maximum. Then, it exists $l \neq j$ such that $\sum_{k=l}^{i} A[k] > \sum_{k=j}^{i} A[k]$.

   $\sum_{k=l}^{i} A[k] > maxending[i-1]+A[i]$ by definition. So $\sum_{k=l}^{i-1} A[k] > maxending[i-1]$ and $maxending[i-1]$ is not the maximum. Contradiction.

3. Compute the value with bottom-up fashion : we compute $maxending[i]$ from $i = 1$ to n, storing the max and the indices. We can notice : $maxending[i-1] + A[i] > A[i] \Leftrightarrow maxending[i-1] > 0$. So we will just look at our local maximum : if it's negative, the new max is $A[i]$.

4. Construct the optimal solution : extract $\max_i maxending[i]$

2.6.2 Complexity of Kadane

We have :

$$\begin{cases} T(n) = T(n-1) + \Theta(1) \\ T(1) = \Theta(1) \end{cases}$$
$$\Leftrightarrow T(n) = \sum_{i=1}^{n} \Theta(1) = \Theta(n)$$

Extract the max is $\Theta(n)$, so the global algorithm is $\Theta(n)$.

2.6.3 Adaptation to 2D array

With $sumrc$, $\forall k, l \in [\![1, m]\!], k < l$, we can compute in $\Theta(n)$ an array where the i^{th} element is the sum of the column from element k to element l.

Then, you can apply kadane's algorithm to find the maximum of this array.

So, in that way, you find the global maximum testing every combination of line indices with the kadane's algorithm, and :

$$\begin{aligned} T(n, m) &= \Theta(m^2) \times (\Theta(n) + \Theta(n)) \\ &= \Theta(m^2 n) \end{aligned}$$

Moreover, you can find the maximum dimension first to obtain $\Theta(\min(m, n)^2 \times \max(m, n))$.

No constrained version for 2D-Kadane Kadane's algorithm is not adapted to find a constrained array. In fact, it's not really usefull such it allready takes a linear time to find a fixed size subarray in an array with the other constrained version of dynamic.

2.7 Randomized approach

A randomized approach introduces an element of randomness, typically by incorporating random choices or perturbations during the computation process. The objective is to diversify the exploration of the solution space and mitigate the risk of getting stuck in local optima. In the randomized approach for the sub-matrix problem, the algorithm incorporate randomization with random initialization.

2.7.1 Unconstrained version

Explanation We start with a given matrix of size (m, n).
The algorithm will first choose random submatrices of maximim size *(m,n)* and minimum size *(1,1)* for a number of trial. For each one, it will then compute the sum of the submatrix, storing it as the best matrix on the first trial. After the first trial, new submatrices will be tested against the so far best sum found, storing it if the current sum is greater than the stored best sum. We also store the line and columns start and end indices of the best submatrix to be able to output our solution.

Pseudo-code

```
random(matrix):
Begin
    Let m,n be the dimensions of the randomly created matrix, with l lines and c columns

    For i <- 1 to m * n
        Submatrix <- randomly selected submatrix
        if i = 1:
            best_score <- sum of current submatrix (l1,c1) to (l2,c2)
            startline,startcolumn,endline,endcolumn <- current indices
        else if submatrix score > best score
            best_score <- sum of current submatrix
            startline,startcolumn,endline,endcolumn <- current indices

    Return startline,startcolumn,endline,endcolumn,best_score, matrix
End
```

Complexity Let $M \in \mathcal{M}_{m,n}(\Re)$ a matrix. The number of operations for *random* is :

$$T(m,n) = \Theta(1) + \sum_{i=1}^{m}\sum_{i=1}^{n} O(mn)$$

$\Theta(1)$ corresponds to the generation of a random submatrix
$\sum_{i=1}^{m}\sum_{i=1}^{n} O(mn)$ corresponds to the computation of the sum of all submatrices generated
In this case, the complexity is:

$$T(m,n) = O(m^2n^2).$$

And for the best case (the randomly selected matrix is a 1*1 matrix) the complexity is :

$$T(m,n) = \Omega(m * n)$$

Conclusion At a first glance, this algorithm is quite simple, but the main question is does it works well ? As a randomized algorithm it's difficult to say yes, indeed it depends quite a lot on luck, on selecting a random matrix that is found to be the best in the whole matrix. Theoretically, it could find the best submatrix if the algorithm was running for an unlimited amount of time, but that would not be realistic. In practice we have to give a certain number of iteration. The result is that on small matrices it has more chance to retrieve the correct solution as the number of possible submatrix is smaller. On bigger matrix it's really unlikely that it would give the best submatrix as the number of submatrix is increasing so much, making it almost impossible to find the correct one randomly. This algorithm is not meant to optimal however it gives a result really fast.

2.7.2 Constrained version

Explanation We start with a given matrix of size (m, n).
The algorithm will first choose random submatrices of size $K*L$ for a number of trial. For each one, it will then compute the sum of this submatrix, storing it as best matrix on the first trial. After the first trial, new submatrices will be tested against the so far best sum found, storing it if the current sum is greater than the stored best sum. We also store the line and columns start and end indices of the best submatrix to be able to output our solution.

Pseudo-code

```
random(matrix):
Begin
    Let m,n be the dimensions of the randomly created matrix, with l lines and c columns

    For i <- 1 to m * n
        Submatrix <- randomly selected submatrix
        If submatrix is of size K*L:
            If i = 1:
                best_score <- sum of current submatrix (l1,c1) to (l2,c2)
                startline,startcolumn,endline,endcolumn <- current indices
            Else if submatrix score > best score
                best_score <- sum of current submatrix
                startline,startcolumn,endline,endcolumn <- current indices

    Return startline,startcolumn,endline,endcolumn,best_score, matrix
End
```

Complexity Let $M \in \mathcal{M}_{m,n}(\Re)$ a matrix. The number of operations for *random* is :

$$T(m,n) = \Theta(m+n) + \Theta(1) + \sum_{i=1}^{m}\sum_{i=1}^{n}\Theta(KL)$$

The second $\Theta(m+n)$ corresponds to the check of the constraint.
$\Theta(1)$ corresponds to the generation of a random submatrix.
$\sum_{i=1}^{m}\sum_{i=1}^{n}\Theta(KL)$ corresponds to the computation of the sum of all submatrices generated
We can prove that :

$$T(m,n) = \Theta(m * n(K * L))$$

.

So $T(m,n) = O(m^2n^2)$ and $T(m,n) = \Omega(mn)$.

Conclusion Like its brother from the unconstrained version, the constrained version of the randomized algorithm almost fall back into the same conclusion, but with higher chance of finding the correct result, in fact, with a constrained the number of submatrices that the algorithm can randomly select decrease drastically, increasing the probabity to find the right answer. But as the unconstrained version, it still has the same flaws of a random algorithm.

2.8 Ant Colony Optimization

Ant colony algorithm is inspired by the behavour of an ant colony finding the shortest path to food. Ants begin to move along all the routes to deliver the food to the anthill. Along the way, they leave a trace of pheromones. On a shorter path, ants will go back and forth more times over the same period, therefore leaving more pheromones on the path. This stonger smell of pheromones attracts ants. Over time, all the ants will switch to the stronger smelling route, and the long paths will be abandoned, as the pheromones on them evaporate. This kind of algorithm works best in finding the optimal path in a graph.

2.8.1 Algorithm

```
ACO(matrix)
Begin
    initialize pheromones
    For every generation do
```

```
        For each ant k do
            construct solution
            evaluate solution
            update the best solution
            End
        update pheromones
        End

    Return best solution indices, sum
End
```

2.8.2 Constructing the Solution

To create the current best solution for each ant, we need to calculate the probability of every possible solution the ant might choose, based on the pheromones and the value of the solution; then choose the solution with the highest probability. At first, the algorithm assigns constant pheromones to every element in the matrix, and later, after each generation of ants have found their solutions, the pheromone traces are updated based on the best solution found. The following is how we calculate the probabilities and update pheromones:

Calculating Probabilities:

$$P_{ij} = \frac{\tau_{ij}^{\alpha}\eta_{ij}^{\beta}}{\sum_{m \in allowed} \tau_{im}^{\alpha}\eta_{im}^{\beta}}$$

P is the probability of the ant choosing a solution, the numerator in this equation shows the desire of the ant to choose this solution, τ being the amount of pheromones on this solution, and η the value of this solution. So, the probability of choosing a solution is proportional to the value of the solution and the amount of Pheromones on it. We divide this amount by the sum of the desires of the ant choosing all the possible solutions. α and β are constants we can change to effect how the algorithm works. Increasing α will result in the pheromones(previous experience) having more effect on the ant's decision, and increasing β will higher the effect of the solution value on the ant's choice.

Updating Pheromones:

- Evaporation:

$$\tau_{ij}(t) = (1 - \rho)\tau_{ij}(t)$$

  ρ is the evaporation rate of the pheromones

- Reinforcement:

$$\Delta\tau_{ij}^{k}(t) = \frac{Q}{L^{k}(t)}$$

  $\Delta\tau$ is the pheromone addition the ant makes on the solution, Q is a constant, and L in the length of the path from the ant to the food, which in this problem is translated to the value of the previous best solution, and we multiply it by Q.

- Pheromone update:

$$\tau_{ij}(t+1) = \tau_{ij}(t) + \sum_{k=1}^{n_k} \Delta\tau_{ij}^{k}(t)$$

  The new pheromones are calculated by adding the amounts of evaporation and reinforcements.

2.8.3 Complexity

- Complexity of the unconstrained version:

  Let M be the matrix of size $(m \times n)$, k the number of ants, and g the number of generations. The time complexity of each function can be computed as:

  initialize$_pheromones$:O(m*n)$probability : The time complexity of the algorithm will be as follows :$

$$T(m,n) = O(g * k * m^2 * n^2)$$

And:

$$k = \sqrt{m * n}$$

Therefore:

$$T(m,n) = O(g\sqrt{mn}m^2n^2)$$

- Complexity of the constrained version:

  In this version, the algorithm should find a subamtrix of size $(K \times L)$, so we do not need to compute the accumulated probabilities of every possible solution for the ant, as there is only one solution with that size for each ant we choose. So the Complexity can be calculated as follows:

$$T(m,n) = O(g\sqrt{mn}KL)$$

2.8.4 Points of Strength and Weakness:

- Strengths:
  - By changing the constants(α,β,Q,ρ, number of ants, and number of generations), we can manipulate the performance of the algorithm to produce the desired outcomes.
  - The accuracy of the algorithm is quite well compared to the non-optimal algorithms, especially for the constrained version.
- Weaknesses:
  - The time complexity for the unconstrained version is simply too high, so this is not a good approach for solving this problem when we are facing big matrices.
  - The unconstrained version does not perform well for matrices of all-negative values.

2.9 Clustering : A personal approach

Clustering is an image processing method used to find objects in an image. Considering that images are just matrices of values for the color of pixels, we can link it to the MMSP. For the case of the MMSP we are searching groups of elements with high values in a matrix in order to create a submatrix with a high sum.

2.9.1 Explanation of the algorithm

The clustering method for the MMSP consist of 3 main steps :

- **Pre-processing**: we do operations over our matrix in order to be more computable for the clustering part
- **Clustering**: we find the group of high values that are neighboring each other and group them in a cluster

- **Cluster selection**: we check the sum of every cluster in order to select the one with the highest value

We will explain each step thoroughly :

- **Pre-processing**

  We have 2 pre-processing methods that we apply in the same time to gain some time complexity :

  - **"Bluring"**: "bluring" the matrix consists in giving an element of the matrix the value of the mean of itself and its neighbors. It helps us having less sparse values and having more consistant clusters.
  - **Thresholding**: : thresholding the matrix consists in changing the values of the matrix to either 1 if they are higher than the mean of the matrix or -1 if they are lower. It gives us a decision matrix with only 1 and -1 as values.

  With the decision matrix that we obtain, we can now search for clusters.

- **Clustering**

  To search for clusters, we put every element with a value of 1 in a list. We then take the first element of this list to create a cluster. We next iterate over the elements of the cluster, if we find a neighbor of one of the element in the list containing all the 1, we put the neighbor in the cluster and eliminate it from the candidate list. After that, we take a new candidate from the list of candidate and do the same operation for a new cluster. The clustering part ends when the candidate list is empty. Finally, we frame the cluster by a matrix. We trim the outer rows and columns that have a negative sum in the decision matrix.

- **Cluster selection**

  Now that we have the indexes of the clusters, we can calculate the sum of all the clusters. We then return the cluster with the highest sum.

2.9.2 Complexity of the clustering method

To calculate the complexity of the whole method, we will divide the problem by calculating the complexity of every step.

- **Pre-processing**

  During the pre-processing, we calculate the mean, we copy a matrix and we blur the element, every step is $\Theta(mn)$ so we have :

$$\begin{aligned} T1(m,n) &= 3\Theta(mn) + \Theta(1) \\ &= \Theta(mn) \end{aligned}$$

- **Clustering**

  During the clustering part, we first fetch all the values with a complexity of $\Theta(mn)$. Then we have the clustering algorithm :

$$\begin{aligned} T2(m,n) &= (\sum_{i=1}^{mn} \sum_{j=1}^{mn-i} O(1)) + O(mn) + O(1) \\ &= \sum_{i=1}^{mn} O(mn-i) \\ &= \sum_{i=0}^{mn-1} O(i) \\ &= O(m^2n^2) \end{aligned}$$

We also can find the lower bound, in this case it's where we have only one cluster that has only one value :

$$\begin{aligned} T2(m,n) &= \Omega(mn) + \Omega(1) \\ &= \Omega(mn) \end{aligned}$$

- **Cluster selection**

  During the cluster selection, we calculate the sum of the matrices framing the clusters. We can study multiple cases since we have two sums that depends on respectively the number of clusters and the max size of a cluster. First we will study the case of the biggest number of clusters, for this we would have only cluster framed by a matrix $F_{1\times1}$. For this disposition to be achieved, we would need that all cluster have no neighbors, so the decision matrix would contain only alternation of 1 and -1. The number of clusters in this matrix would be of $\frac{m\times n}{2}$. In this case, since the sum of the frame would be $O(1)$, we would have a complexity of $O(\frac{m\times n}{2})$

  The second possible case is one cluster framed by a matrix $F_{m\times n}$. In this case we would compute only one time the sum of the matrix F, so we would have a complexity of $O(mn)$. Our worst case would be a matrix $W_m \times m$ that looks like this (here for $m = 5$ and the -1 are replaced by 0 for readability purpose) :

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

  It is the matrix with the biggest number of clusters containing a cluster framed by the entire matrix. In this case we would have have a complexity of :

$$\begin{aligned} T3(m,n) &= \sum_{i=1}^{\frac{mn}{2}} O(1) + O(mn) \\ &= O(\frac{mn}{2}) + O(mn) \\ &= O(mn) \end{aligned}$$

  So we can conclude that $T3(m,n) = O(mn)$ For the best case, we only have one cluster that is a matrix with one element, in this case we have a complexity of $T3 = \Omega(1)$

We can now conclude that our complexity for the approach in the worst case is :

$$\begin{aligned} T(m,n) &= 2O(mn) + O(m^2n^2) \\ &= O(m^2n^2) \end{aligned}$$

And for the best case the complexity is :

$$\begin{aligned} T(m,n) &= 2\Omega(1) + \Omega(mn) \\ &= \Omega(mn) \end{aligned}$$

2.9.3 Criticism on the method

The clustering method is a method that is great at giving a "good" result. It will often not find a solution that is optimal but will always find a solution that is close to it while being quicker than other methods. It is however slow compared to the greedy approach or the randomized approach. It is also as slow as the dynamic approach while the dynamic approach can give the optimal solution.

3 Code library

Every programms explained in the Theoretical Framework section 2 has been code in Python3. You can find the library attached with this document.

3.1 How to use the library

1. Place the `submax` folder in the same repertory as your `file.py`
2. Add the command line `import submax` at the begining of your file.

- Every not constrained function will be import as `submax.method.solve()`. The only parameter is a `np.ndarray` type 2D-matrix.

  `method` cans be : "`ant`, `bandb` (brunch and bound), `brute` (bruteforce), `divandconq` (divide and conquer), `dynamic`, `greedy`, `random`".

  For bruteforce, there is `solve_c` and `solve_lm`, because there is two methods. Default function `solve` is c version.

- Every constrained function will be import as `submax.method.constraint()`. Parameters are `(2D-matrix, number of line (K), number of column (L))`. Please notice *there is not constrained version of the divide and conquer approach.*
- All programms return the indexes of the maximum sub-array of the form `(startline, startcolumn, endline, endcolumn)`, and the `maximum` sum corresponding.
- From `niceprint`, there is `submax.printsol()`, a function which hilight a submatrix in a global matrix (see the subsection below). It takes 2 parameters : the global matrix, and the indexes of the subamtrix of the form `(startline, startcolumn, endline, endcolumn)`.

3.2 Visualize the solutions

In the goal of debbuging and understanding our solutions in a simple way, we design a simple algorithm to visualize without to many effort the outputted solutions. The idea is to used the colorama library of Python to highlight a desired submatrix in a global matrix.

The code can be found and used in the `submax` library.

This function was created at the begining of our project because it has seemed required to be efficient to find if the algorithms was correct and, more important, if they were not, why so.

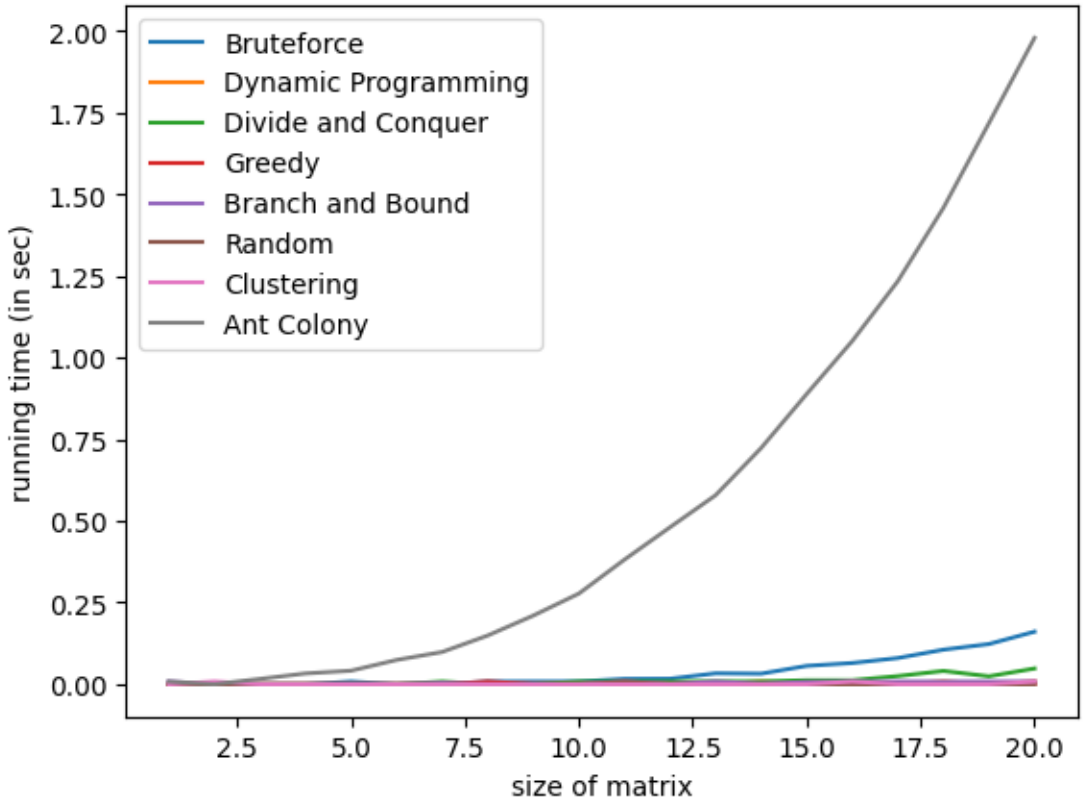
4 Benchmark

4.1 Unconstrained Version

Using different sizes and unique matrices it is interesting to compare all algorithms, how fast they are going in real time, and how well they do to find the best submatrix.

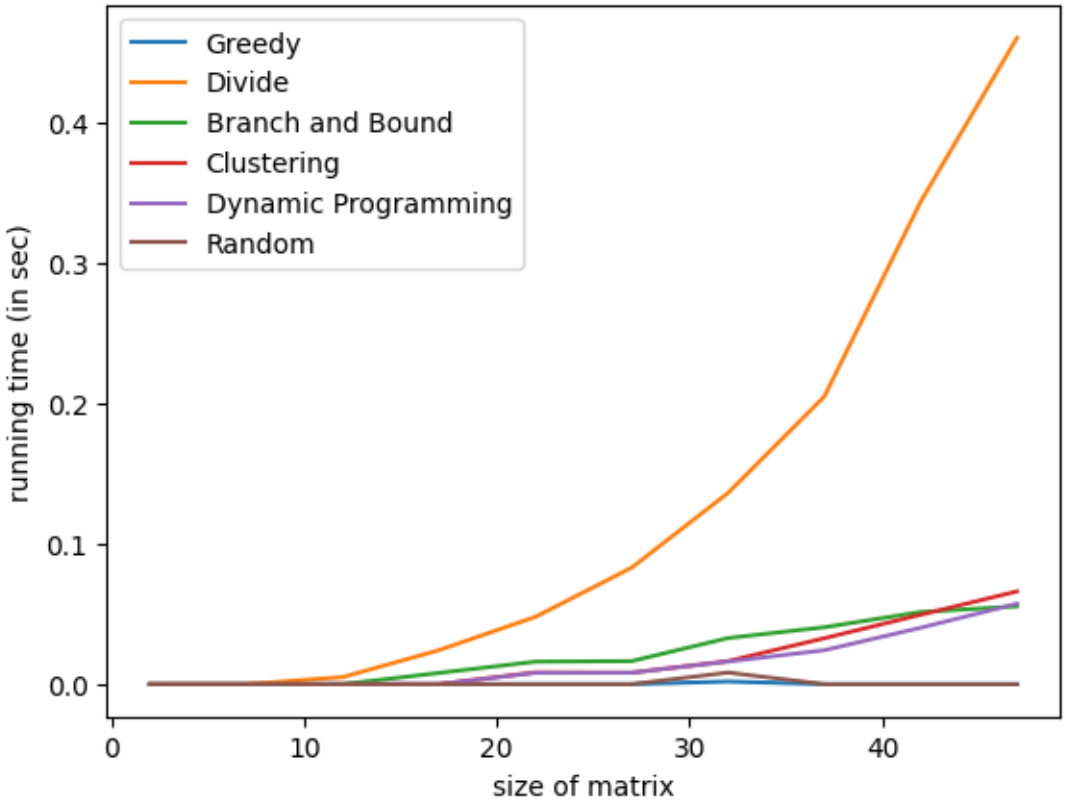
4.1.1 Running Time

Using 20 square matrices, with increasing size (from 1x1 to 20x20) and random values between -100 and 100, we can observe how much time the algorithm takes to be completed.



On this graph, we easily see two algorithms being slower than the rest. Not surprisingly, the bruteforce algorithm takes a lot of time as it computes everything before returning its solution. But the Ant Colony algorithm is much slower and it seems that it requires a lot of resources. After that, we see that the Divide and Conquer method is slower than the rest but still doing a better job than the bruteforce. As we can't really say anything about the other algorithms, we could remove the bruteforce and Ant Colony ones to be able to see more precisely how fast the others are.

As these algorithms are faster, we increased the sizes of the matrices, using 10 matrices from size 1x1 to 50x50 with a step of 5 and still using values from -100 to 100.



We still clearly observe that the Divide and Conquer method is slower, but this graph is helping us see how fast the Clustering, Dynamic Programming and Branch and Bound are, as they all to seem similar. Compared to the previous algorithms they work much faster. Finally, the Randomized and the greedy method are working the fastest, finishing almost instantly.

However, the running time doesn't tell us how good the algorithms are at retrieving the maximum submatrix.

4.1.2 Algorithm accuracy

To test our algorithms, we will compare the results with the bruteforce results, as it computes all submatrices, we know it gives back the correct result all the time (even though it is slow).

The test will be done on the 20 matrices of increasing size used in the first running-time test. Additionally, we add a list of specific test matrices : The project matrix, big, medium, and small matrices, positive and negative matrices, column and line matrices, a matrix with a column as solution, and another one with a line. In total, we have 30 matrices

We then compute the number of times the algorithms don't have the same result as bruteforce (Errors) and also the total difference in their sums compared to the best submatrix found (Total Difference). The result obtained are on the following table

	Errors	Total Difference
Dynamic Programming	0	0
Divide and Conquer	0	0
Greedy	21	16067
Branch and Bound	14	4823
Randomized	28	14751
Clustering	21	9307
Ant Colony	19	3977

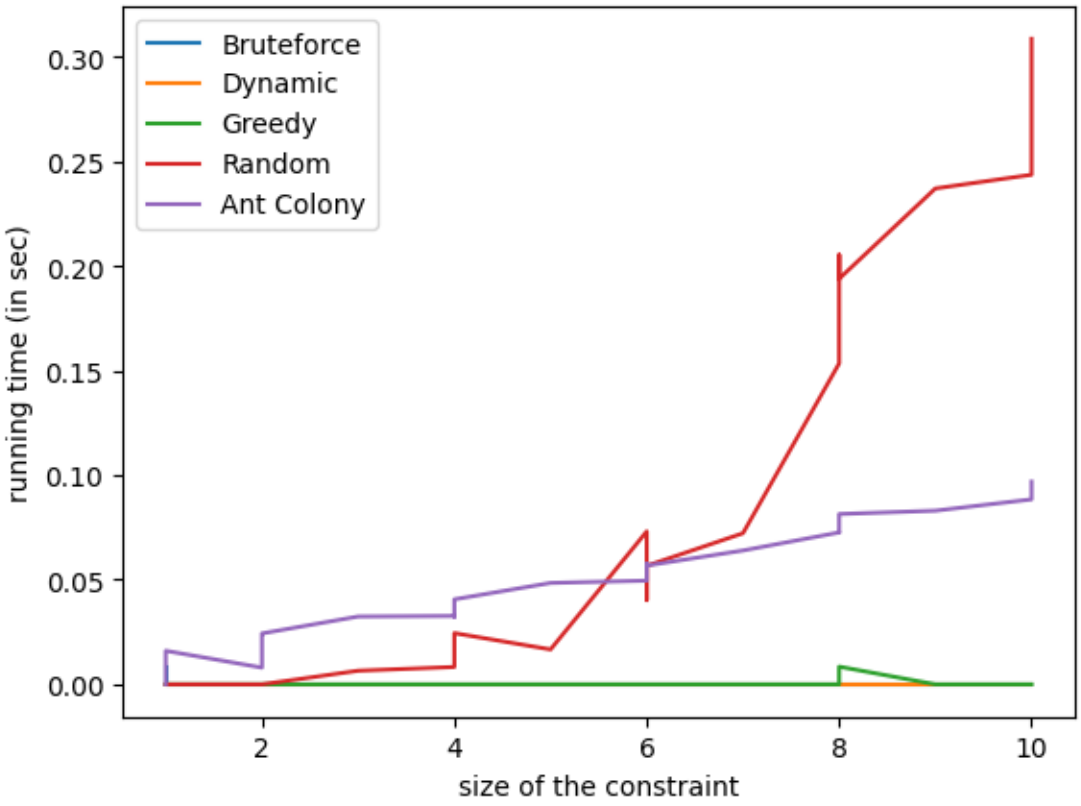
With this, we see that 2 of the algorithms are actually retrieving optimal solutions on all the matrices used : Dynamic Programming and Divide and Conquer.

For the rest, they mostly don't get the correct submatrix, however the Ant Colony algorithm is closer in general to the optimal solution. Coming after, The Branch and Bound and the Clustering algorithm are also doing a better job than other non-optimal algorithms. Greedy and Randomized are making more mistakes and fail to be close to the solutions found by the bruteforce algorithm.

4.2 Constrained Version

4.2.1 Running Time

To compute the running time on our constrained versions (bruteforce, dynamic, greedy, randomized, and ant colony), we use the same matrices that we used previously (of increasing size from 1x1 to 20x20) with a constraint of half it's size, meaning K and L would be 10 if the matrix is a 20x20 matrix.



We can see that the constraint version running time is quite different compared to the unconstrained version. It makes sense as the algorithms all have different complexity. We see here that Bruteforce and Greedy are running really fast while the Randomized approach is a lot slower than before. Additionally, we find that Ant Colony works a lot faster in its constrained version. The Branch and Bound version is not plotted, because this version works in exponential time, making it really slow.

4.2.2 Algorithm accuracy

As done for the unconstrained version, we check the results all algorithms with the results retrieved by the bruteforce algorithm. We don't use the Branch and Bound algorithm, because of it has a really slow computation.

	Errors	Total Difference
Dynamic Programming	0	0
Greedy	21	9076
Ant Colony	0	0
Randomized	20	5158

On the different constraint version we see that Dynamic Programming is still retrieving all correct submatrices, more surprisingly the Ant Colony approach is, on all test matrices, finding all the correct solutions; While the Randomized approach and the greedy one are still not getting the same submatrices as the bruteforce approach most of the time. It is also surprising that the Randomized approach is closer to the actual solutions than the greedy one.

5 Conclusion

- Despite of some algorithms having similar time complexity, it was observed that the time of execution of these aproaches can be very distant. This is because of the nature of each aproach, how it explores the seach space and how it takes into account possible best solutions. This means that there are more factors other than time complexity to evaluate the performance of an aproach.
- Some aproaches are suposed to find the optimal solution
- As shown in the plots and tables, out of all the algortihms we worked on, Dynamic programming seems to be the best approach for solving this problem, as it always produces the optimal solution, while having a relatively small runtime in both the unconstrained and constrained versions. Both Bruteforce and Divide and Conquer, despite calculating the optimal solutions in the unconstrained problem, are not efficient, as they take a comparatively long time to produce the results. Greedy, Randomized and Clustering algorithms are also not the best approaches for unconstrained, since they produce a lot of errors, in spite of having a good runtime.

A Weekly progress

- 27/10/2023

  As a team we started to review the project in order to split the task effectively, and made sure everyone understood the project. We created deadline on a Trello and a GitHub repository. Charles, Louis and Mohadeseh created a bruteforce approach. Charles calculated its space complexity. Ugo created a Randomized approach.

- 10/11/2023

  Charles created a print function to show the solution retrieved by algorithms, created pseudo code for its bruteforce approach and studied the dynamic approach. Louis created a greedy approach. Ugo finished its code for the Randomized approach, included a pseudo code and the complexity. Mohadeseh started some research on Genetic and Ant Colony algorithms. Alejandro worked on the Branch and Bound approach.

- 17/11/2023

  Charles did a Dynamic Programming algorithms, including a pseudo code and python code. Louis finshed the greedy approach and its constrained version and explained the theory if it. Ugo finished the constrained version of the Randomized algorithms and its complexity. Alejandro finished its Branch and Bound approach and came up with its complexity, he also worked on the constrained version.

- 24/11/2023

  Charles correct bugs on different function, implemented Kadane's algorithms and studied its complexity. He also created a library to unify all approach. Louis studied the time complexity for greedy and

improved his approach, he created examples matrices to test the different algorithms on. Ugo started working on a Benchmark notebook with comparison on running time and the sum of error of each algorithms. Mohadeseh studied Genetic and Ant Colony algorithms and started the initial version of the Ant Colony approach. Alejandro worked on the constrained version and its complexity.

- 01/12/2023

  Charles created a Divide and Conquer approach, commented all his code and wrote about strictly bruteforce theory. Louis created a Clustering approach, studied its complexity and tested it. Ugo added new accuracy features to the Benchmark, implemented new algorithms on it and started writing the benchmark and Randomized approach part on the report. Mohadeseh created her Ant Colony algorithms and fixed bugs on it. Alejandro studied the complexity of the constrained version of Branch and Bound, worked on a constrained version of dynamic Programming and tested algorithms.

B Workload

- Alejandro : –%. Tasks done : Branch and bound (uncontrained and constrained version), Dynamic programming (Constrained with the help of Charles)
- Charles : 10π%. Tasks done : Bruteforce (unconstrained), Bruteforce constrained theory, Divide and Conquer (unconstrained), Dynamic programming (unconstrained), 2D-Kadane (unconstrained), printing function, library assembling.
- Louis : 200%. Tasks done : Greedy approach (unconstrained and constrained version), Clustering approach
- Mohadeseh : –%. Tasks done : Bruteforce (unconstrained), Ant Colony approach (unconstrained and constrained version)
- Ugo : –%. Tasks done : Randomized approach (unconstrained and constrained version), Benchmark

C Checklist

1. Did you proofread your report?
2. Did you present the global objective of your work?
3. Did you present the principles of all the methods/algorithms used in your project?
4. Did you cite correctly the references to the methods/algorithms that are not from your own? Did you include all the details of your experimental setup to reproduce the experimental results, and explain the choice of the parameters taken?
5. Did you provide curves, numerical results and error bars when results are run multiple times?
6. Did you comment and interpret the different results presented?
7. Did you include all the data, code, installation and running instructions needed to reproduce the results?
8. Did you engineer the code of all the programs in a unified way to facilitate the addition of new methods/techniques and debugging?
9. Did you make sure that the results different experiments and programs are comparable?
10. Did you comment your code sufficiently?
11. Did you add a thorough documentation on the code provided?
12. Did you provide the additional planning and the final planning in the report and discuss organization aspects in your work?

13. Did you provide the workload percentage between the members of the group in the report?
14. Did you send the work in time?