Advanced Algorithm Project 2023-2024

Comparing the Complexity of Various Algorithms Used to Solve Maximum sum rectangle in a 2D matrix

by

Meryem Ben Yahia
Amgad Khalil
Anna Karolin
Amrithya Balaji
Youssouf Ali Ahmed

Université Jean Monnet
Saint-Étienne

Contents

1 Introduction 3

2 Problem Description 3
2.1 Maximum sum rectangle in a 2D matrix . . . 3
2.1.1 Constrained Formulation . . . 4
2.2 Project and Data Description . . . 4

3 Software Used 5

4 Algorithms 6
4.1 Brute Force . . . 6
4.1.1 Application Of Brute Force on the problem . . . 6
4.1.2 Algorithm . . . 7
4.1.3 Complexity Analysis . . . 7
4.2 Dynamic Programming . . . 8
4.2.1 Approach for Max sum sub-matrix . . . 8
4.2.2 Algorithm . . . 9
4.2.3 Complexity Analysis . . . 10
4.3 Greedy Approach . . . 11
4.3.1 Experimental Greedy . . . 11
4.3.2 Approach for Max sum sub-matrix Using a modified greedy algorithm 12
4.3.3 Algorithm . . . 12
4.3.4 Complexity Analysis . . . 13
4.4 Branch and Bound . . . 13
4.4.1 First Implementation . . . 14
4.4.2 Complexity Analysis . . . 16
4.4.3 Second Implementation . . . 17
4.4.4 Complexity Analysis . . . 20
4.5 Randomized Version . . . 20
4.5.1 Randomized Constrained version . . . 21
4.5.2 Complexity Analysis . . . 22
4.5.3 Randomized Unconstrained version . . . 24
4.5.4 Complexity Analysis . . . 25
4.6 Genetic Algorithm (GA) . . . 27
4.6.1 Principles of Genetic Algorithm . . . 27
4.6.2 Genetic Algorithm: Constrained Variant . . . 27
4.6.3 Complexity Analysis of Constrained GA . . . 29
4.6.4 Genetic Algorithm: Unconstrained Variant . . . 30
4.6.5 Complexity Analysis of Unconstrained GA . . . 33
4.7 Ant Colony Optimization (ACO) . . . 33
4.7.1 No Constraint Variant . . . 33
4.7.2 Complexity Analysis of UnConstrained ACO . . . 36
4.7.3 Constrained Variant . . . 37
4.7.4 Complexity Analysis of Constrained ACO . . . 39
4.7.5 Study . . . 39
4.7.6 Experimental Setup and Results . . . 41
4.7.7 Comparative Analysis and Conclusion . . . 42

5 Summary and conclusions 42

6 Distribution of Work 48

7 Adherence to Provisional Planning **49**

8 Acknowledgements **49**

9 Checklist: Questions and Answers **50**

Advanced Algorithm Project 2023-2024

Comparing the Complexity of Various Algorithms Used to Solve Maximum sum rectangle in a 2D matrix

Abstract

Efficiently solving the Maximum Sum Rectangle problem in a 2D matrix is a critical task in various computational applications, ranging from image processing to genome sequencing. This project presents a comprehensive comparative analysis of the computational complexity of multiple algorithms designed to address this challenge. Our study focuses on evaluating the efficiency and performance of algorithms such as brute-force, dynamic programming,branch and bound, randomized, ant-colony, genetic, greedy approaches.

Keywords: Kadane's algorithm, Brute-Force, Dynamic Programming, Branch and Bound, Randomized, Ant-Colony, Genetic, Greedy

1. Introduction

The Maximum Sum Rectangle problem, as identified in the literature[Wik], presents a significant algorithmic challenge with diverse applications across various fields. This problem, situated within the broader class of maximum sub-array problems, is of great importance due to its practical implications in domains such as image processing, data analysis, genomics sequence analysis, and computer vision. Genomics sequence analysis employs these algorithms to identify important biological segments of protein sequences. These problems include conserved segments, GC-rich regions, tandem repeats, low-complexity filters, DNA binding domains, and regions of high charge.
In this report, we explore various algorithmic approaches to solve the maximum sum rectangle problem in a 2D matrix. The aim was to identify algorithms that not only offer theoretical efficiency but also align well with practical constraints. The project entails the implementation, evaluation, and comparative analysis of multiple algorithmic approaches, including brute-force, dynamic programming, branch-and-bound, greedy, randomized, and genetic programming/ant colony methodologies, to address the Matrix Maximum Segment problem in a 2D scenario.

2. Problem Description

2.1. ***Maximum sum rectangle in a 2D matrix***

The objective is to implement algorithms using different for solving the Matrix maximum segment problem in the two-dimensional scenario - known also as the Maximum sub-array Problem - and to provide an experimental study of their running time and quality. The Maximum segment sum problem, is the task of finding a contiguous segment with the largest sum, within a given two-dimensional array.

Given a 2-D array $A = (a_{i,j})_{1 \le i \le M, 1 \le j \le N}$ of size $M \times N$ with integers, we need to find the indices i_1, i_2, j_1, j_2 where $1 \le i_1 \le i_2 \le M$, $1 \le j_1 \le j_2 \le N$ such that the sum

$$\sum_{x=i_1}^{i_2} \sum_{y=j_1}^{j_2} A[x, y]$$

1	2	-1	-4	-20
-8	-3	4	2	1
3	8	10	1	3
-4	-1	1	7	-6

Figure 1: Maximun sub-matrix from codingninjas

is as large as possible.

2.1.1. ***Constrained Formulation***

Given the same 2-D array A of size $M \times N$ with integers, and the shape constraints K, L, we need to find the indices i_1, i_2, j_1, j_2 where $1 \leq i_1 \leq i_2 \leq M$, $1 \leq j_1 \leq j_2 \leq N$, $i_2 - i_1 + 1 = K$, $j_2 - j_1 + 1 = L$ such that the sum

$$\sum_{x=i_1}^{i_2} \sum_{y=j_1}^{j_2} A[x, y]$$

is as large as possible. In simple terms, we need to find the sub-array of size $K \times L$ with the maximum sum possible.

2.2. ***Project and Data Description***

The project is hosted and version controlled in GitHub and the link for GitHub repository. The execution of the project is simple.

1. The necessary libraries in Python required to run the algorithms are numpy, timeit, seaborn, matplotlib, collections, random and unittest.
2. Each algorithm has a separate folder and a main file. The folder contains the functions and its helper functions of the respective algorithms. The main file imports all the necessary functions and the test class to run the program.
3. The test class evaluates each algorithm based on the test cases which is described below and gives success and failure percentages of the algorithm. The test class also plots a pie and bar chart for success and failure percentage and the graph for run time vs the size of the matrix.
4. The main_all file calls all the functions of the algorithm specifically to plot the comparison graphs in the Summary and Conclusions section.

We have three sets of matrices. The details include matrices with specific values, expected results, sub-array indices, and constraints. The matrices are randomly generated by a python code snippet using NumPy. The function runs through all test matrices against an existing working solution to find the expected results and appends it to the test case. The test cases are described below.

Test Scenario: Non-Constrained Test Cases

- Matrix Sizes: Randomly generated matrices with sizes ranging from small to large of size $M \times N$, with M and N positive integers such that $2 \leq M \leq 22$ and $2 \leq N \leq 22$.
- Matrix Values: Random values in the range of -100 to 100.

- The primary metric for evaluation is the run time of the algorithm concerning the size of the input matrices.
- The test case contains the matrix, the expected max-sum of the sub-array, and the indices of the sub-matrix.

Conclusions are drawn based on the observed performance of the algorithm with non-constrained test cases.

Test Scenario: Constrained Fixed Test Cases

We created matrices with increasing sizes while keeping constraints K and L fixed. This scenario aims to evaluate the algorithm's performance when matrix sizes vary, but the constraints remain constant.

- Matrix Sizes: Randomly generated matrices with sizes ranging from small to large of size $M \times N$, with M and N positive integers such that $2 \leq M \leq 22$ and $2 \leq N \leq 22$.
- Matrix Values: Random values in the range of -100 to 100.
- Constraints: Fixed values for K and L throughout the test.
- The primary metric for evaluation is the run-time of the algorithm concerning the size of the input matrices.
- The test case contains the matrix, the expected max-sum of the sub-array, the indices of the sub-matrix, and the constraint.

Draw conclusions based on the observed performance of the algorithm with constrained fixed test cases.

Test Scenario: Variable Constrained Test Cases

We utilized a constant 15×15 matrix with values ranging from -100 to 100. The objective is to test the algorithm's performance with the same matrix under different constraints K and L.

- Matrix Size: 15×15 matrix.
- Matrix Values: Random values in the range of -100 to 100.
- Constraints: Different values for K and L to assess the algorithm's adaptability.
- The primary metric for evaluation is the run-time of the algorithm concerning the constraints applied.
- The test case contains the matrix, the expected max-sum of the sub-array, the indices of the sub-matrix, and the constraint.

Draw conclusions based on the observed performance of the algorithm with variable-constrained test cases.

3. Software Used

- GitHub is used for version control and collaborative software development. It provides a central repository for code hosting, version tracking, and project management.
- Jupyter Notebook, Google Colab, and Visual Studio Code are widely utilized for interactive computing, as they allow for the combination of code, documentation, and visualizations within a single document. These platforms support various programming languages, making them ideal for tasks such as data analysis and prototyping.

4. Algorithms

In this section, we describe each of the algorithms we implemented and provide pseudo-code and the computation complexity of each algorithm used.

4.1. ***Brute Force***

The brute-force search method is a straightforward approach for problem-solving in computer science. This algorithm iterates over all possible solutions for a given problem. It checks whether each solution satisfies the problem's goals and constraints. Despite being easy to implement, the algorithm's time complexity grows quickly with respect to the number of possible solutions. This method cannot be used for solving many real-world problems because they have a large solution set. However, brute-force search can typically be used in problems that have a limited size[gee].

4.1.1. ***Application Of Brute Force on the problem***

- **Unconstrained Version**
  In the context of finding the maximum sum sub-array in a 2D matrix, the brute force approach involves fixing a top left corner calculating the sum of every possible sub-array for this fixed corner, and then determining which one has the highest sum. This approach is exhaustive and considers all potential sub-arrays within the matrix.

- **Constrained Version**
  The constrained brute force approach is similar to the unconstrained version but with an additional check before calculating the sum for if the constraints ($K \times L$) rows and columns respectively are within the boundaries.

4.1.2. ***Algorithm***

Algorithm 1 Brute Force Maximum Sum Submatrix No Constraints

```
function CalculateSubarraySum(matrix, top, left, bottom, right)
    subarray_sum ← 0
    for i ← top to bottom do
        for j ← left to right do
            subarray_sum ← subarray_sum + matrix[i][j]
        end for
    end for
    return subarray_sum
end function
function BruteForceNonConstrained(matrix)
    rows ← number of rows in matrix
    cols ← number of columns in matrix
    max_sum ← −∞
    top_left ← (0, 0)
    bottom_right ← (0, 0)
    for top ← 0 to rows − 1 do
        for left ← 0 to cols − 1 do
            for bottom ← top to rows − 1 do
                for right ← left to cols − 1 do
                    current_sum ← CalculateSubarraySum(matrix, top, left, bottom, right)
                    if current_sum > max_sum then
                        max_sum ← current_sum
                        top_left ← (top, left)
                        bottom_right ← (bottom, right)
                    end if
                end for
            end for
        end for
    end for
    return (max_sum, top_left, bottom_right)
end function
```

4.1.3. ***Complexity Analysis***

Let M and N represent the number of rows and columns of the matrix, respectively.

Time Complexity Analysis

The time complexity of the algorithm is analyzed as follows:

- The algorithm iterates through all possible sub-matrices. This involves four nested loops:
  - The first two loops (for `top` and `left`) iterate over all possible top-left corners, with $M \times N$ iterations.
  - The next two loops (for `bottom` and `right`) iterate over all possible bottom-right corners for each top-left corner, again with $M \times N$ iterations in the worst case.
  - Therefore, the combined complexity for these four loops is $O(M^2 \times N^2)$.
- Within the innermost loop, the `calculate_subarray_sum` function is called, which iterates over the elements of a submatrix. In the worst case, it runs $M \times N$ times for each submatrix.
- Thus, the total time complexity of the algorithm is $O(M^2 \times N^2 \times M \times N) = O(M^3 \times N^3)$.

Space Complexity Analysis

The space complexity of the algorithm is analyzed as follows:

- The algorithm uses a constant amount of extra space for storing variables such as the maximum sum, indices of the submatrix, and the current sum.
- Therefore, the space complexity of the algorithm is $O(1)$, independent of the size of the input matrix.

4.2. ***Dynamic Programming***

Dynamic programming (DP) is a method for efficiently solving a broad range of search and optimization problems that exhibit the property of overlapping sub-problems and optimal substructure. It involves breaking down a problem into smaller, overlapping sub-problems and solving each sub-problem only once, storing the solutions to sub-problems in a table to avoid redundant computations. Dynamic programming is typically used for optimization problems, where the goal is to find the best solution among a set of feasible solutions. To solve optimization problems by DP, two properties are necessary:

- An optimal solution can be decomposed into optimal solutions of sub-problems. This property can be shown by using proofs by contradiction.
- The sub-problems must overlap, i.e. be called many times to solve the original problem

4.2.1. ***Approach for Max sum sub-matrix***

The use of cumulative sums[Ped] helps to efficiently calculate the sum of any sub-matrix in constant time by combining the sums of previously calculated sub-matrices.

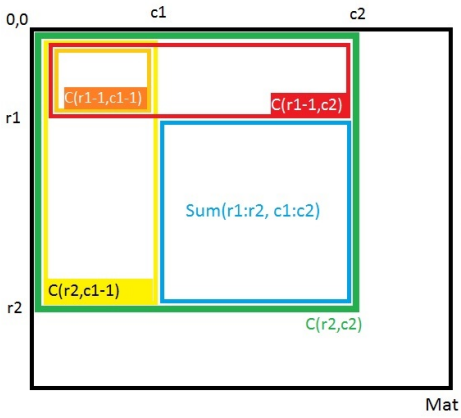


Figure 2: Calculating Cumulative sum of the matrix

Non Constrained version.

- Firstly we calculate the cumulative sums of the input matrix. This involves creating a new matrix 'cumulative_sums' where 'cumulative_sums[i][j]' represents the sum of all elements in the sub-matrix from $(0, 0)$ to $(i - 1, j - 1)$ in the original matrix.
- After computing the cumulative sums, iterate over all possible sub-matrices and calculate their sum using the cumulative sums matrix. The sub-matrix sum is calculated using the cumulative sums matrix to subtract the sums of the appropriate bordering sub-matrices.
- Keep track of the maximum sum encountered so far and the corresponding indices (top, left, bottom, right) that define the boundaries of this sub-matrix.

Constrained version.

- Iteration have been modified to start iterating the sub-matrices from K and L instead of 1. This is because you are now considering sub-matrices of size K rows by L columns.
- The sub-matrix sum is still calculated using the cumulative sums matrix. The only change is in the indices used for the cumulative sums to get the sum of the sub-matrix with dimensions K by L.
- The indices (top, left, bottom, right) are adjusted based on the new loop ranges and the dimensions $K \times L$.

4.2.2. ***Algorithm***

Algorithm 2 DP Maximum Sum Sub-matrix No Constraints

```
1: function maxMatrixSum_non_constraint(matrix)
2:     rows ← len(matrix)
3:     cols ← len(matrix[0])
4:     c ← zeros(rows + 1, cols + 1)
5:     for i ← 1 to rows do
6:         for j ← 1 to cols do
7:             c[i][j] ← matrix[i − 1][j − 1] + c[i − 1][j] + c[i][j − 1] − c[i − 1][j − 1]
8:         end for
9:     end for
10:    max_sum ← −∞
11:    top, left, bottom, right ← 0, 0, 0, 0
12:    for i1 ← 1 to rows do
13:        for j1 ← 1 to cols do
14:            for i2 ← i1 to rows do
15:                for j2 ← j1 to cols do
16:                    submatrix_sum ← c[i2][j2] − c[i1 − 1][j2] − c[i2][j1 − 1] + c[i1 − 1][j1 − 1]
17:                    if submatrix_sum > max_sum then
18:                        max_sum ← submatrix_sum
19:                        top ← i1 − 1
20:                        left ← j1 − 1
21:                        bottom ← i2 − 1
22:                        right ← j2 − 1
23:                        top_left ← (top, left)
24:                        bottom_right ← (bottom, right)
25:                    end if
26:                end for
27:            end for
28:        end for
29:    end for
30:    return max_sum, top_left, bottom_right
31: end function
```

Algorithm 3 DP Maximum Sum Sub-matrix Constraints

```
1: function maxMatrixSum_constraint(matrix, K, L)
2:     rows ← len(matrix)
3:     cols ← len(matrix[0])
4:     c[] ← zeros(rows + 1, cols + 1)
5:     for i ← 1 to rows do
6:         for j ← 1 to cols do
7:             c[i][j] ← matrix[i − 1][j − 1] + c[i − 1][j] + c[i][j − 1] − c[i − 1][j − 1]
8:         end for
9:     end for
```

Algorithm 4 DP Maximum Sum Sub-matrix Constraints - Part 2

```
function maxMatrixSum_constraint(matrix, K, L)                    ▷ Continued
    max_sum ← −∞
    top, left, bottom, right ← 0, 0, 0, 0
    for i ← K to rows do
        for j ← L to cols do
            submatrix_sum ← c[i][j] − c[i − K][j] − c[i][j − L] + c[i − K][j − L]
            if submatrix_sum > max_sum then
                max_sum ← submatrix_sum
                top ← i − K
                bottom ← i − 1
                left ← j − L
                right ← j − 1
            end if
        end for
    end for
    return max_sum, (top, left), (bottom, right)
end function=0
```

4.2.3. ***Complexity Analysis***

Let M and N represent the number of rows and columns of the matrix, respectively.

The **space complexity** of the algorithm in both non-constrained and constrained version is same and is analyzed as follows:

- A 2D array (cumulative_sums) is created to store cumulative sums. The size of this array is $O((M+1) \times (N+1))$, which requires $O(M \times N)$ space.
- A few integer variables (max_sum, top, left, bottom, right) are used, which require constant space.
- The overall space complexity is dominated by the cumulative sums calculation, so the total space complexity is $O(M \times N)$ space.

The **time complexity** of the algorithm is analyzed as follows:

Non constrained version.

- Two nested loops are used to calculate cumulative sums with $M \times N$ iterations, which takes $O(M \times N)$. time.
- Four nested loops are used for calculating sub-matrix sums. The limits of these loops depend on the size of the matrix. The time complexity for the sub-matrix sums calculation is $O(M^2 \times N^2)$ because there are two nested loops iterating over the rows and two nested loops iterating over the columns.
- The overall time complexity is dominated by the sub-matrix sums calculation, so the total time complexity is $O(M^2 \times N^2)$

Constrained version.

- Two nested loops are used to calculate cumulative sums with $M \times N$ iterations, which takes $O(M \times N)$. time.
- Four nested loops are used for calculating sub-matrix sums. The limits of these loops depend on the size of the matrix. The time complexity for the sub-matrix sums calculation is $O((M-K+1) \times (N-L+1))$ as these loops iterate over a smaller portion of the matrix.

- The overall time complexity is dominated by the sub-matrix sums calculation, so the total time complexity is $O(M \times N)$

4.3. ***Greedy Approach***

A greedy algorithm is any algorithm that follows the strategy of choosing the choice that looks best at the moment with the hope of finding the global optimum, called a locally optimal choice. Sometimes, this strategy will lead to a globally optimal solution but this is not always guaranteed. One advantage of such algorithms is that they eliminate a lot of the steps required for solving a problem therefore making it more efficient.

4.3.1. ***Experimental Greedy***

We experimented with the following approach but ended up not using it because it was failing 98 percent of the time against our test cases shown in fig: 3. This greedy approach selects the best sub-matrix at each step. First, we start by iterating over the entire matrix to find the single element with maximum value. Second, expand the current sub-matrix to include adjacent elements rows, and columns. This expansion is considered in four directions (up, down, left, right), effectively creating four candidate sub-matrices. For each of the candidates, we calculate their sum. We then choose the sub-matrix that gives the best sum at this stage and iterate over it. The iteration stops once all the candidates do not have a better sum than the current sub-matrix sum. Here are some of the cases where it fails to get an optimal solution:-

- if the algorithm expands the sub-matrix in a direction that initially appears beneficial but misses a larger sum available in another part of the matrix
- if the maximum value in the matrix is surrounded by negative numbers only.

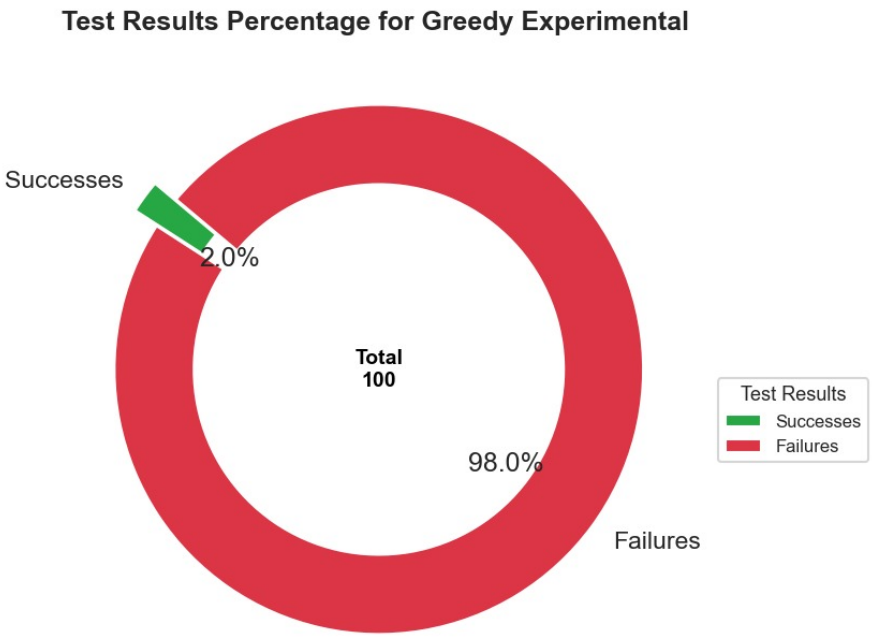


Figure 3: Greedy Experimental success rate

4.3.2. ***Approach for Max sum sub-matrix Using a modified greedy algorithm Non Constrained version.***

- The greedy_algorithm_non_constrained function is designed to identify the sub-matrix with the maximum sum within a given matrix, employing a greedy algorithm with a dynamic programming component. The algorithm begins by initializing the maximum sum (max_sum) to negative infinity and the coordinates of the sub-matrix (start_row, end_row, start_col, end_col) to -1.
- Subsequently, an outer loop is established to iterate over all potential left columns (left) from 0 to cols - 1. Within this loop, a temporary array (temp) is initialized with zeros to store the running sum of rows. The algorithm proceeds to a middle loop, iterating over possible right columns (right) starting from the current left column.
- For each combination of left and right columns, an inner loop accumulates the running sum in the temporary array for all rows (i). The values in temp are updated with the cumulative sum from the left column to the current right column.
- max_subarray_sum_1d_with_indices function utilizes Kadane's algorithm [kad], to find the maximum sub-array sum in the 1D array temp. This involves maintaining a running sum (current_sum) and tracking the start and end indices of the maximum sub-array. The running sum is updated based on whether extending the current sub-array or starting a new sub-array would yield a larger sum.
- The maximum sub-array sum and its corresponding coordinates are updated if the current sub-array sum (current_sum) surpasses the current max_sum. Finally, the function returns the overall maximum sum and the coordinates of the sub-matrix.

4.3.3. ***Algorithm***

Algorithm 5 max_subarray_sum_1d_with_indices (kadane's algorithm)

```
function MAX_SUBARRAY_SUM_1D_WITH_INDICES(arr)
    max_sum ← −∞
    current_sum ← 0
    current_start ← temp_start ← temp_end ← 0
    for i, num in enumerate(arr) do
        if current_sum + num < num then
            current_sum ← num
            current_start ← i
        else
            current_sum ← current_sum + num
        end if
        if current_sum > max_sum then
            max_sum ← current_sum
            temp_start ← current_start
            temp_end ← i
        end if
    end for
    return max_sum, temp_start, temp_end
end function
```

Algorithm 6 greedy_algorithm_non_constrained

```
function GREEDY_ALGORITHM_NON_CONSTRAINED(matrix)
    rows ← len(matrix)
    cols ← len(matrix[0])                                   ▷ Assuming non-empty matrix
    max_sum ← −∞
    start_row ← end_row ← start_col ← end_col ← −1
    for left ← 0 to cols − 1 do
        temp ← [0] ∗ rows
        for right ← left to cols − 1 do
            for i ← 0 to rows − 1 do
                temp[i] ← temp[i] + matrix[i][right]
            end for
            current_sum, current_start_row, current_end_row ← max_subarray_sum_1d_with_indices(temp)
            if current_sum > max_sum then
                max_sum ← current_sum
                start_row, end_row, start_col, end_col ← current_start_row, current_end_row, left, right
            end if
        end for
    end for
    return max_sum, (start_row, start_col), (end_row, end_col)
end function
```

4.3.4. ***Complexity Analysis***

Let M and N represent the number of rows and columns of the matrix, respectively.
The **space complexity** of the algorithm is analyzed as follows:

- A 1D array (temp) is created to store sums. The size of this array is M, which requires $O(M)$ space.
- A few integer variables (max_sum, top, left, bottom, right) are used, which require constant space.
- The overall space complexity is dominated by the calculation of the cumulative sums, so the total space complexity is $O(M)$.

The **time complexity** of the algorithm is analyzed as follows:

- The max_subarray_sum_1d_with_indices function, based on Kadane's algorithm with N,the number of columns has a time complexity of $O(N)$.
- In Greedy Algorithm function, the outer loop iterates over all possible left columns with a complexity of $O(N)$.The middle loop iterates over all possible right columns with a complexity of $O(N)$. The inner loop iterates over all rows, accumulating the temporary array of right columns with a complexity of $O(M)$.
- The overall time complexity is dominated by the calculation of the sub-matrix sum, so the total time complexity is $O(M \times N^2)$

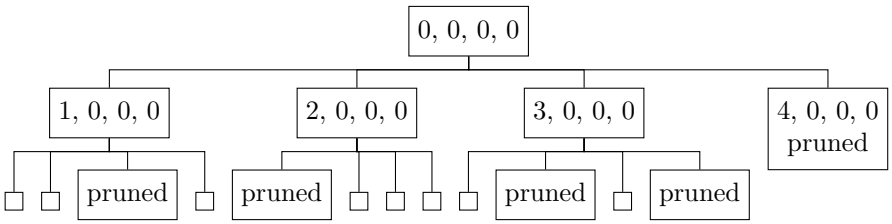
4.4. ***Branch and Bound***

The branch and bound approach divides a problem into several sub-problems that can be solved recursively and bounding with a function so as to not compute sub cases which are not optimal. This approach was first devised by Land and Doig[Mor+16] in 1960. The approach consists of algorithms that follows the general rule of dividing into sub-problems(branching) and the use of a function to prune off sub-optimal cases(bounding). The time complexity of this approach greatly depend on the implementation of the algorithm as we can use this technique on various algorithms to reduce their complexity. In the worst case scenario we will be looking at all the sub-cases which will give us a time complexity which is exponential.

Here two implementations using the branch and bound approach has been done.

4.4.1. ***First Implementation***

- **Initialization** : We transform the matrix of size $M \times N$ into a tree to carry out a breadth first search and initialize the maximum sum to negative infinity and maximum sub-matrix to none.
- **Working** : We take the root node to be an array of size 4, add it to a queue. Then we pop the root from the queue and generate m child nodes for this node by changing the value of 1st element of the array i^{th} child node to i where $i \in \{1, 2, \ldots, M\}$ and calculate the sum and the sum of the positive elements for each child node. We add a child node to the queue if the sum of positives elements of the node is greater than the maximum sum.
- **Updating Rule** : We update the maximum sum and maximum sub-matrix to sum of child node and indices of child node if the sum of elements of the child node is greater than maximum sum. We prune off the nodes whose sum of positive elements is less than maximum sum.
- **Continuing** : We follow the same procedure for updating the second element and for the third and fourth elements, we take N instead of M. We continue the procedure till the queue is empty. Then we output the maximum sum and maximum sub-matrix(Algorithm 9).



Algorithm 7 Helper Functions for Max Segment Branch and Bound (Part 1)

```
function VALID(child, m, n)
    if child[1] = 0 then
        child[1] ← m
    end if
    if child[3] = 0 then
        child[3] ← n
    end if
    if (child[0] < child[1]) or (child[2] < child[3]) then
        return True
    end if
end function
function GENERATECHILDREN(node, m, n)
    children ← []
    for i from 0 to 3 do
        if node[i] = 0 then
        end if
        for j from 1 to (m if i < 2 else n) do
            child ← node.copy()
            child[i] ← j
            if VALID(child, m, n) then
                children.append(child)
            end if
        end for

    end for
    return children
end function
function ADJUST(child, matrix)
    child[0] ← 1 if child[0] = 0 else child[0]
    child[1] ← matrix.shape[0] if child[1] = 0 else child[1]
    child[2] ← 1 if child[2] = 0 else child[2]
    child[3] ← matrix.shape[1] if child[3] = 0 else child[3]
    return child
end function
```

Algorithm 8 Helper Functions for Max Segment Branch and Bound (Part 2)

```
function BOUND(child, matrix)
    i1, i2, j1, j2 ← child
    i1 ← 1 if i1 = 0 else i1
    i2 ← matrix.shape[0] if i2 = 0 else i2
    j1 ← 1 if j1 = 0 else j1
    j2 ← matrix.shape[1] if j2 = 0 else j2
    sub ← matrix[i1 − 1 : i2, j1 − 1 : j2]
    positive_values ← sub[sub > 0]
    up_bound ← Sum(positive_values)
    child_sum ← Sum(sub)
    return (up_bound, child_sum)
end function
```

Algorithm 9 Max Segment Branch and Bound

```
1: function MAX_SEGMENT_BRANCH_AND_BOUND(matrix)
2:     matrix ← ConvertToMatrix(matrix)
3:     m, n ← matrix.shape
4:     max_sum ← −∞
5:     max_sum_submatrix ← None
6:     indices ← [0, 0, 0, 0]
7:     q ← Queue()
8:     q.Enqueue(indices)
9:     n_iter ← 4
10:    for i from 0 to n_iter do
11:        while not q.IsEmpty() do
12:            node ← q.Dequeue()
13:            for child in GENERATE_CHILDREN(node, m, n) do
14:                up_bound, child_sum ← bound(child, matrix)
15:                if up_bound ≥ max_sum then
16:                    q.Enqueue(child)
17:                    if child_sum > max_sum then
18:                        max_sum ← child_sum
19:                        max_sum_submatrix ← child
20:                    end if
21:                end if
22:            end for
23:        end while
24:    end for
25: end function
```

4.4.2. ***Complexity Analysis***

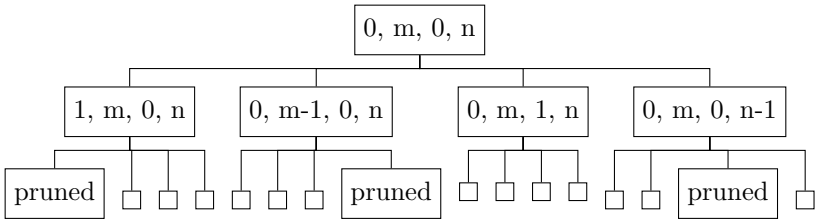
Space Complexity. In the worst case, we will be considering the whole tree which will be $\mathcal{O}(m^2 * n^2)$.

Time Complexity. The execution of the while loop in the worst case generates the whole tree which will be of complexity $\mathcal{O}(m^2 * n^2)$ and the complexity for the function *generating_children* is $\mathcal{O}$(max(m, n)). Thus the time complexity of the algorithm comes out to be $\mathcal{O}((m^2 * n^2)$*(max(m, n))).

4.4.3. ***Second Implementation***

- **Initialization** : Here we transform the matrix of size $M \times N$ into a tree and carry out a breadth first search and initialize the maximum sum to the sum of the matrix and maximum sub-matrix to the whole matrix.
- **Working** : We consider the whole matrix to be the root node and add it to a queue. We pop the first element of the queue and then similar to a brute force way we explore the sub-matrices of the matrix (generate child nodes) but we don't explore the sub-matrices whose indices are one plus or minus that of the parent node, if possible. We prune of child nodes whose sum of positive elements are less than the maximum sum.
- **Updating Rule** : We add those child nodes into the queue whose sum of positive elements is higher than the maximum sum and we update the values of maximum sum and maximum sub-matrix whose child node sum of elements is greater than maximum sum.
- **Continuing** :. We continue the procedure till the queue is empty. Then we output the maximum sum and maximum sub-matrix(Algorithm 11).

In the constrained version we add an additional check if the constrains are satisfied before updating the values of maximum sum and maximum sub-matrix(Algorithm 12).



Algorithm 10 Helper Function for Max Segment Branch and Bound

```
1: function PositiveSum(matrix)
2:     s ← 0
3:     for row in matrix do
4:         for i in row do
5:             if i > 0 then
6:                 s ← s + i
7:             end if
8:         end for
9:     end for
10:    return s
11: end function
```

Algorithm 11 Max Segment Branch and Bound Non Constrained

```
function MaxSegmentBranchAndBound(matrix)
    matrix ← np.array(matrix)
    m ← matrix.shape[0]
    n ← matrix.shape[1]
    initial_partial ← ((0, m − 1, 0, n − 1), np.sum(matrix))
    initial_best ← PositiveSum(matrix)
    function Valid(i1, i2, j1, j2)
        return i1 ≤ i2 and j1 ≤ j2 and i1 ≥ 0 and i2 < m and j1 ≥ 0 and j2 < n
    end function
    function MatrixSubSum(i1, i2, j1, j2)
        return np.sum(matrix[i1:i2+1, j1:j2+1])
    end function
    function PositiveSubSum(i1, i2, j1, j2)
        s ← 0
        for i in range(i1, i2 + 1) do
            for j in range(j1, j2 + 1) do
                if matrix[i, j] > 0 then
                    s ← s + matrix[i, j]
                end if
            end for
        end for
        return s
    end function
    function GenerateChildren(partial, best)
        i1, i2, j1, j2 ← partial[0]
        children ← []
        if Valid(i1 + 1, i2, j1, j2) then
            children.append((i1 + 1, i2, j1, j2))
        end if
        if Valid(i1, i2 − 1, j1, j2) then
            children.append((i1, i2 − 1, j1, j2))
        end if
        if Valid(i1, i2, j1 + 1, j2) then
            children.append((i1, i2, j1 + 1, j2))
        end if
        if Valid(i1, i2, j1, j2 − 1) then
            children.append((i1, i2, j1, j2 − 1))
        end if
        return children
    end function
    q ← deque([(initial_partial, initial_best)])
    d ← {initial_partial[0] : (initial_partial[1], initial_best)}
    while q do
        current_partial, current_best ← q.popleft()
        for child in GenerateChildren(current_partial, current_best) do
            if child ∉ d.keys() then
                a ← MatrixSubSum(child)
                b ← PositiveSubSum(child)
                d[child] ← (a, b)
            end if
            if d[child][1] > initial_partial[1] then
                q.append((child, d[child][0]))
                if d[child][0] > initial_partial[1] then
                    initial_partial ← (child, d[child][0])
                end if
            end if
        end for
    end while
    return initial_partial[1], (initial_partial[0][0], initial_partial[0][2]), (initial_partial[0][1], initial_partial[0][3])
end function
```

Algorithm 12 Max Segment Branch and Bound Constraint

```
function MaxSegmentBranchAndBoundConstraint(matrix, k, l)
    m ← matrix.shape[0]
    n ← matrix.shape[1]
    initial_partial ← ((0, m − 1, 0, n − 1), np.sum(matrix))
    initial_best ← PositiveSum(matrix)
    function Valid(i1, i2, j1, j2)
        return i1 ≤ i2 and j1 ≤ j2 and i1 ≥ 0 and i2 < m and j1 ≥ 0 and j2 < n
    end function
    function MatrixSubSum(i1, i2, j1, j2)
        return np.sum(matrix[i1:i2+1, j1:j2+1])
    end function
    function PositiveSubSum(i1, i2, j1, j2)
        s ← 0
        for i in range(i1, i2 + 1) do
            for j in range(j1, j2 + 1) do
                if matrix[i, j] > 0 then
                    s ← s + matrix[i, j]
                end if
            end for
        end for
        return s
    end function
    function GenerateChildren(partial, best)
        i1, i2, j1, j2 ← partial[0]
        children ← []
        if Valid(i1 + 1, i2, j1, j2) then
            children.append((i1 + 1, i2, j1, j2))
        end if
        if Valid(i1, i2 − 1, j1, j2) then
            children.append((i1, i2 − 1, j1, j2))
        end if
        if Valid(i1, i2, j1 + 1, j2) then
            children.append((i1, i2, j1 + 1, j2))
        end if
        if Valid(i1, i2, j1, j2 − 1) then
            children.append((i1, i2, j1, j2 − 1))
        end if
        return children
    end function
    q ← deque([(initial_partial, initial_best)])
    d ← {initial_partial[0] : (initial_partial[1], initial_best)}
    while q do
        current_partial, current_best ← q.popleft()
        for child in GenerateChildren(current_partial, current_best) do
            if child ∉ d.keys() then
                a ← MatrixSubSum(child)
                b ← PositiveSubSum(child)
                d[child] ← (a, b)
            end if
            if d[child][1] > initial_partial[1] then
                q.append((child, d[child][0]))
                if d[child][0] > initial_partial[1] and (child[1] − child[0] + 1 == k) and (child[3] −
child[2] + 1 == l) then
                    initial_partial ← (child, d[child][0])
                end if
            end if
        end for
    end while
    return initial_partial[1], (initial_partial[0][0], initial_partial[0][2]), (initial_partial[0][1], initial_partial[0][3])
end function
```

4.4.4. ***Complexity Analysis***

The following applies for both the non constrained and constrained versions of second implementation.

Space Complexity. In the worst case, we will be considering the whole tree whose depth will be max(m, n). We also have a branching factor, which id the number of child nodes generated for each node which is 4. Thus the space complexity come out to be $\mathcal{O}(max(m, n))$.

Time Complexity. The time complexity also depends on the branching factor and the depth of the tree and comes out to be $\mathcal{O}(4^{(max(m,n))})$ which is exponential.

4.5. ***Randomized Version***

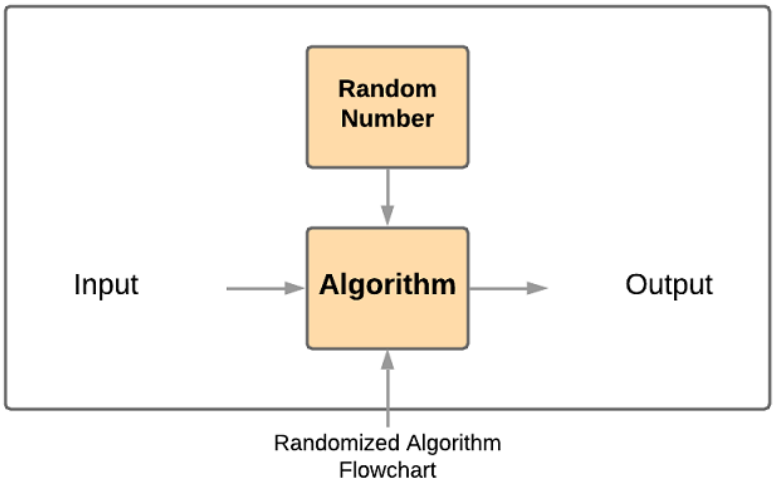


Figure 4: Simplified Randomized Algorithm Flowchart

A randomized algorithm is an algorithm that receives, in addition to its input data, a stream of random bits used to make random choices. Even for a fixed input, different runs of a randomized algorithm may give different results; thus, it is inevitable that a description of the properties of a randomized algorithm will involve probabilistic statements [Kar91].

In the context of solving the Matrix Maximum Segment Problem, randomized algorithms offer distinct advantages. One of the key strengths of such algorithms lies in their efficiency in exploring the solution space, particularly beneficial in large matrices. Unlike deterministic approaches that methodically examine each segment, randomized algorithms randomly select segments for evaluation. This randomness enables more rapid discovery of optimal or near-optimal solutions, especially valuable in scenarios where deterministic methods may be overwhelmed by the numerous possible segments.

In the case of worst-case complexity, randomized approaches can mitigate extreme scenarios. In the unconstrained version of the problem, where brute-force deterministic methods exhibit high computational complexity, a randomized algorithm can circumvent this by skipping certain segments. In the constrained formulation, where the task is to find a sub-array of a specific size ($K \times L$), randomized algorithms demonstrate a significant advantage. By randomly selecting starting points and sizes within the specified constraints, a randomized algorithm can uncover high-sum segments in an unbiased way compared to systematic, deterministic exploration. Additionally, randomized algorithms are generally less susceptible to biases induced by distributions or patterns within the input matrix.

4.5.1. ***Randomized Constrained version***

Pseudo-code of Algorithm

Algorithm 13 Random Fixed Size Submatrix Sum

```
function RandomFixedSizeSubmatrixSum(matrix, K, L)
    M ← number of rows in matrix
    N ← number of columns in matrix
    num_iterations ← int((M × (M + 1) × N × (N + 1))/4)
    max_sum ← −∞
    best_submatrix ← None
    for iteration ← 1 to num_iterations do
        i1 ← random integer between 0 and M − K
        j1 ← random integer between 0 and N − L
        i2 ← i1 + K − 1
        j2 ← j1 + L − 1
        current_sum ← 0
        for i ← i1 to i2 do
            for j ← j1 to j2 do
                current_sum ← current_sum + matrix[i][j]
            end for
        end for
        if current_sum > max_sum then
            max_sum ← current_sum
            best_submatrix ← ((i1, j1), (i2, j2))
        end if
    end for
    return max_sum, best_submatrix[0], best_submatrix[1]
end function
```

1. **Initialization:**
   - **M and N**: The function starts by determining the dimensions of the input matrix. M is the number of rows and N is the number of columns in the matrix.
   - **num_iterations**: It calculates the number of iterations for the algorithm to run. The formula `int((M*(M+1)*N*(N+1))/4)` is used to determine this number, which is based on the dimensions of the matrix.
   - **max_sum**: Initialized to negative infinity (`float('-inf')`) to ensure that any sum found in the matrix will be larger.
2. **Randomized Submatrix Selection:**
   - The function enters a loop that runs for `num_iterations` times.
   - In each iteration, it randomly selects a starting point for a submatrix. $i1$ and $j1$ are the starting row and column indices, chosen randomly such that a full $K \times L$ submatrix fits within the bounds of the larger matrix of size $M \times N$.
   - $i2$ and $j2$ are calculated to be the ending row and column indices of the submatrix, ensuring it has the correct size of $K \times L$.
3. **Calculating the Sum of the Submatrix:**
   - `current_sum` is calculated as the sum of elements within the selected submatrix. This is done using a nested loop that iterates over each element in the submatrix and adds its value to `current_sum`.
4. **Updating the Maximum Sum and Best Submatrix:**
   - After calculating the sum of the current submatrix, the function checks if this sum is greater than the current `max_sum`.

- If `current_sum` is larger, it updates `max_sum` with this new value and updates `best_submatrix` with the current submatrix's coordinates.

5. **Returning the Result:**
   - After completing all iterations, the function returns three values: the maximum sum found (`max_sum`), the starting indices of the submatrix with this sum (`best_submatrix[0]`), and the ending indices of this submatrix (`best_submatrix[1]`).

*4.5.2. **Complexity Analysis***

Time Complexity

1. **Initialization**:
   - Calculating M and N, the dimensions of the matrix, is $O(1)$, as it's a direct attribute access.
   - Calculating `num_iterations` is also $O(1)$, being a simple arithmetic operation based on M and N.
2. **Main Loop**:
   - The algorithm runs for `num_iterations` times, where `num_iterations` is set to $\frac{M(M+1)N(N+1)}{4}$, which is approximately $O(M^2N^2)$ for large values of M and N.
   - Inside each iteration:
     - Generating random indices $i1$ and $j1$ is $O(1)$.
     - Calculating the sum of elements in the submatrix defined by $i1$, $j1$, $i2$, and $j2$ has a complexity of $O(K \times L)$, where K and L are the dimensions of the submatrix because it iterates over every element in the $K \times L$ submatrix.
3. **Summation of Complexities**:
   - The dominant part of the algorithm is the nested loop for summing the elements in the submatrix, repeated `num_iterations` times. Therefore, the overall time complexity of the algorithm is $O(M^2N^2KL)$.

Space Complexity

1. **Stored Variables**:
   - The algorithm uses a fixed number of variables (M, N, `num_iterations`, max_sum, $best\_submatrix$, $i1$, $i2$, $j1$, $j2$, $current\_sum$). Each of these requires $O(1)$ space complexity.
2. **Input Matrix**:
   - The input matrix, of size $M \times N$, is a significant part of the space requirement. However, as this matrix is an input to the algorithm and not created within it, its space is often considered external to the algorithm's space complexity.
3. **Temporary Variables**:
   - The use of temporary variables in loops also incurs a constant amount of space.

Therefore, the space complexity of the algorithm is $O(1)$, indicating that it requires a constant amount of space regardless of the size of the input.

Analysis and Evaluation

The number of iterations chosen corresponds to the number of all possible submatrices in a matrix of dimension $M \times N$ [tan].

- The number of elements in a list of length m is $\frac{m(m+1)}{2}$. Also, an $m \times 1$ matrix can be viewed as a list of length m, and the submatrices of that matrix are the sublists of the given list.

- Therefore, we expect the formula for the number of submatrices to reduce to $\frac{m(m+1)}{2}$ when $n = 1$. Intuitively, we can guess that the number of submatrices should at least be proportional to the square of m and n, i.e., $f(m, n) = \Omega(m^2n^2)$, where $f(m, n)$ is the value we are seeking, i.e., the number of submatrices of an $m \times n$ matrix.

To obtain the total number of possible submatrices:

- **Horizontal boundaries**: We form a submatrix with $m + 1$ horizontal lines. This can be done in $\binom{m+1}{2}$ ways.
- **Vertical boundaries**: Similarly, we form a submatrix with $n + 1$ vertical lines. This selection can be done in $\binom{n+1}{2}$ ways.

Thus, the total number of submatrices $f(m, n)$ can be determined by multiplying these two independent choices:

$$f(m, n) = \binom{m+1}{2} \times \binom{n+1}{2}$$

Given an algorithm that performs a number of trials equal to the total number of possible submatrices in an input matrix, while only searching for submatrices of size $K \times L$, we aim to calculate the probability of finding both the right optimal maximum sum and the global optimal maximum sum submatrix.

The following is the probability calculation:

- The total number of submatrices in an $m \times n$ matrix is given by:

$$\frac{m(m+1)n(n+1)}{4}$$

- However, the number of $K \times L$ submatrices in an $m \times n$ matrix is:

$$(m - K + 1)(n - L + 1)$$

- Probability of Selecting a Specific $K \times L$ Submatrix in One Trial:

$$\frac{1}{(m - K + 1)(n - L + 1)}$$

- Probability of Never Selecting the Optimal $K \times L$ Submatrix in One Trial:

$$1 - \frac{1}{(m - K + 1)(n - L + 1)}$$

- Probability of Never Selecting the Optimal $K \times L$ Submatrix in All Trials:

$$\left(1 - \frac{1}{(m - K + 1)(n - L + 1)}\right)^{\frac{m(m+1)n(n+1)}{4}}$$

- Probability of Selecting the Optimal $K \times L$ Submatrix At Least Once:

$$1 - \left(1 - \frac{1}{(m - K + 1)(n - L + 1)}\right)^{\frac{m(m+1)n(n+1)}{4}}$$

The probability of the algorithm finding the optimal $K \times L$ submatrix at least once in all trials is high, especially if the number of trials equals the total number of possible submatrices in the input matrix.

4.5.3. ***Randomized Unconstrained version***

Pseudo-code of Algorithm

Algorithm 14 Randomized Gradient Descent

```
function CALCULATE_ITERATIONS(M, N)
    c ← 0.5
    b ← 100
    iterations ← int(c × (M + N) + b)
    return max(iterations, b)
end function
function CALCULATE_TRIALS(M, N)
    b ← 100
    trials ← (M × N) + b
    return max(trials, b)
end function
function CREATE_CUMULATIVE_SUM_MATRIX(matrix)
    cum_sum_matrix ← pad matrix with zero padding on the top and left
    cum_sum_matrix[1 :, 1 :] ← cumulative sum of matrix along rows and columns
    return cum_sum_matrix
end function
function SUBMATRIX_SUM(cum_sum_matrix, top_left, bottom_right)
    tl_x, tl_y ← top_left
    br_x, br_y ← bottom_right
    return cum_sum_matrix[br_x + 1, br_y + 1] − cum_sum_matrix[tl_x, br_y + 1]
        −cum_sum_matrix[br_x + 1, tl_y] + cum_sum_matrix[tl_x, tl_y]
end function
function GRADIENT_DESCENT_SUBMATRIX(cum_sum_matrix, M, N)
    max_iterations ← CALCULATE_ITERATIONS(M, N)
    max_sum ← −∞
    best_submatrix ← ((0, 0), (0, 0))
    Randomly initialize top_left and bottom_right within the matrix dimensions
    for _ in range max_iterations do
        current_sum ← SUBMATRIX_SUM(cum_sum_matrix, top_left, bottom_right)
        if current_sum > max_sum then
            max_sum ← current_sum
            best_submatrix ← (top_left, bottom_right)
        end if
        [Perform adjustments to top_left and bottom_right]
        [Update top_left and bottom_right if improvement found]
    end for
    return max_sum, best_submatrix
end function
function FIND_MAX_SUM_SUBMATRIX(matrix)
    M, N ← dimensions of matrix
    num_trials ← CALCULATE_TRIALS(M, N)
    overall_max_sum ← −∞
    overall_best_submatrix ← ((0, 0), (0, 0))
    cum_sum_matrix ← CREATE_CUMULATIVE_SUM_MATRIX(matrix)
    for _ in range num_trials do
        max_sum, best_submatrix ← GRADIENT_DESCENT_SUBMATRIX(cum_sum_matrix, M, N)
        if max_sum > overall_max_sum then
            overall_max_sum ← max_sum
            overall_best_submatrix ← best_submatrix
        end if
    end for
    return overall_max_sum, overall_best_submatrix[0], overall_best_submatrix[1]
end function
```

1. **Function** `calculate_iterations`
    - The function calculates the number of iterations for the gradient descent-like method. It

takes the dimensions of the matrix (M and N) as inputs and determines the number of iterations based on the formula $c \times (M + N) + b$, where c is a constant (0.5) and b is a base number of iterations (100).

2. **Function** `calculate_trials`
   - The function computes the number of trials for searching the maximum sum submatrix. The number of trials is based on matrix size ($M \times N$) plus a base number (b).
3. **Function** `create_cumulative_sum_matrix`
   - The function creates a cumulative sum matrix from the input matrix. It pads the input matrix with zeros on the top and left, and calculates the cumulative sum row-wise and column-wise.
4. **Function** `submatrix_sum`
   - The function calculates the sum of elements in a submatrix using the cumulative sum matrix using the top-left and bottom-right coordinates of the submatrix.
5. **Function** `gradient_descent_submatrix`
   - The function applies a gradient descent-like method to find the submatrix with the maximum sum. It randomly initializes a submatrix and iteratively improves it, adjusts the submatrix (expands, shrinks, moves) to find a better sum, and returns the maximum sum and coordinates of the best submatrix.
6. **Function** `find_max_sum_submatrix`
   - The function repeatedly applies `gradient_descent_submatrix` in each trial with the objective of determining the overall best submatrix across all trials and returns it.

*4.5.4. **Complexity Analysis***

Time Complexity

- The most computationally intensive part of the algorithm is the `find_max_sum_submatrix` function.
- The overall time complexity of the entire algorithm is approximately $O(MN \times ((M + N)))$.

Space Complexity

1. **Input Matrix**
   - `create_cumulative_sum_matrix` **Function**:
     - Creates a cumulative sum matrix (`cum_sum_matrix`) that is slightly larger than the input matrix due to padding.
     - Space Complexity: $O(MN)$.
2. **Temporary Variables**
   - Temporary variables in functions like `submatrix_sum` and `gradient_descent_submatrix` use a constant amount of space.
   - Space Complexity: $O(1)$.
3. **Summation of Complexities**
   - The overall space complexity of the algorithm is dominated by the space required for the cumulative sum matrix, which is $O(MN)$.
   - The other parts of the algorithm contribute only a constant amount of additional space.

Analysis and Evaluation

Gradient descent is a method for refining variables within a function to find its local minimum by systematically progressing toward the point of greatest negative gradient. In Machine Learning, gradient descent iteratively adjusts a model's parameters to minimize the cost function, thus enhancing the model's predictive accuracy[ml-].

The function `gradient_descent_submatrix` is a heuristic optimization algorithm that performs a task analogous to gradient descent. Unlike the traditional gradient descent which makes continuous parameter adjustments according to the gradient of the cost function, this function employs discrete operations, such as expanding, shrinking, or moving the submatrix, to incrementally find one with the optimal element sum until no better adjustment is possible. Thus, the gradient descent-like approach used in the algorithm does not guarantee finding the global maximum sum submatrix but increases its likelihood by iterative exploration.

To estimate the probability of the algorithm finding the global maximum sum submatrix of the input matrix, we assume that:

- **Uniform Distribution**: Values in the matrix are uniformly distributed.
- **Independence**: Each selection of a starting submatrix is independent.
- **Fixed Number of Trials**: Let T be the total number of trials, and I the number of iterations per trial.
- **Matrix Characteristics**: For a matrix of size $M \times N$, the total number of $K \times L$ submatrices is $S = (M - K + 1)(N - L + 1)$.

1. **Probability of Selecting a Specific Submatrix in One Trial**:
$$\frac{1}{S}$$
2. **Probability of Finding the Global Maximum in One Trial**:
$$\text{Simplified Upper Bound} = I \times \frac{1}{S}$$
3. **Probability of Finding the Global Maximum in All Trials**:
   - Probability of *not* finding the global maximum in one trial:
   $$1 - I \times \frac{1}{S}$$
   - Probability of *not* finding the global maximum in all T trials:
   $$\left(1 - I \times \frac{1}{S}\right)^T$$
   - Therefore, the probability of finding the global maximum in at least one of the trials is:
   $$1 - \left(1 - I \times \frac{1}{S}\right)^T$$

For smaller matrices or when the submatrices are small (in which S is small), the probability of finding the global maximum increases, making this algorithm most effective for applications with small matrices. In the cases of large matrices, the number of potential submatrices (S) becomes very large, reducing the probability of finding the global maximum. Additionally, when computational resources are limited or there are strict time constraints, running a high number of iterations or trials is not feasible, and thus the algorithm is not suitable.

4.6. ***Genetic Algorithm (GA)***

The Genetic Algorithm (GA) [ga]is a bio-inspired computational method designed to solve complex problems by mimicking the process of natural selection. In our project, GA has been expertly adapted to tackle the Matrix Maximum Segment Problem in both its unconstrained and constrained forms. This problem challenges us to identify the contiguous subarray within a two-dimensional numerical matrix that achieves the highest sum of elements. The versatility of GA allows us to explore a wide array of potential solutions, catering to both scenarios where the size of the segment is either fixed or variable.

4.6.1. ***Principles of Genetic Algorithm***

The fundamental principle of GA is rooted in the concept of natural selection and genetic inheritance. It operates on a population of potential solutions to a given problem, employing the process of selection, crossover (recombination), and mutation to generate new offspring. This approach is particularly relevant to the Matrix Maximum Segment Problem, where each potential solution or 'individual' within the GA population represents a specific segment within the matrix.

Unconstrained Variant

In the unconstrained variant of GA, we allow the algorithm to explore the entire solution space, comprising every possible segment within the matrix. This approach is essential when there are no pre-defined constraints on the segment size or shape, thus enabling a comprehensive search across all potential sub-arrays.

Constrained Variant

In contrast, the constrained variant of GA restricts the solution space to segments that conform to pre-specified dimensions, denoted as $K \times L$. This variant is particularly advantageous in situations where the size of the segment is crucial or determined by external factors. Tailoring the GA to these constraints involves modifying genetic operations such as selection, crossover, and mutation to ensure that they operate within the set boundaries.

Both variants of the GA demonstrate the algorithm's adaptability and robustness in solving complex optimization problems, particularly in analyzing and processing large data sets in the form of matrices.

4.6.2. ***Genetic Algorithm: Constrained Variant***

Overview

In the constrained variant of the Genetic Algorithm (GA), our focus is on solving the Matrix Maximum Segment Problem, where we search for the sub-array with the maximum sum, restricted by specific dimensions $K \times L$. This approach introduces a unique challenge: navigating a solution space that only includes segments of a predefined size.

Algorithmic Adaptations for Constraints

To effectively implement this constrained GA, we made several key modifications:

Individual Representation. Each individual in the GA population represents a sub-array of the matrix. The individual's genetic structure encodes the top-left corner coordinates (i, j) of the sub-array. Therefore, each individual can be represented as:

$$\text{Individual} = (i, j)$$

where $1 \leq i \leq M - K + 1$ and $1 \leq j \leq N - L + 1$.

Fitness Function. The fitness function evaluates the suitability of a sub-array as a solution. It calculates the sum of the elements within the sub-array defined by the individual's genetic encoding. The fitness function, f, for an individual is given by:

$$f(i,j) = \sum_{x=i}^{i+K-1} \sum_{y=j}^{j+L-1} A[x,y]$$

where A is the matrix.

Selection Process. Selection is based on the fitness of individuals. Higher fitness values indicate better solutions. The probability of an individual being selected for reproduction, P(Individual), is proportional to its fitness.

Crossover and Mutation. Crossover and mutation are critical for introducing variability and enabling exploration within the solution space.

- **Crossover**: Combines the genetic material of two parent individuals to produce new offspring. If parents $p_1 = (i_1, j_1)$ and $p_2 = (i_2, j_2)$ are selected, the offspring might inherit the row index from one parent and the column index from the other.
- **Mutation**: Introduces random changes in the offspring's genetic makeup, ensuring diversity. This might involve randomly altering the row or column index within the valid range.

Implementation and Iterative Process

The GA operates over multiple generations. In each generation:

1. Individuals are selected based on their fitness.
2. Crossover and mutation are applied to create a new generation.
3. The new generation replaces the old one, and the process repeats.

Algorithm 15 Genetic Algorithm with Constraint - Part 1: Supporting Functions

```
import random
random.seed(42)                                          ▷ Fixed seed for reproducibility

function CALCULATE_SUBARRAY_SUM(matrix, top, left, K, L)
                                                         ▷ Calculate sum of KxL subarray from (top, left)
    sum = 0
    for i in range(top, min(top + K, len(matrix))) do
        for j in range(left, min(left + L, len(matrix[0]))) do
            sum += matrix[i][j]
        end for
    end for
    return sum
end function

function CREATE_INDIVIDUAL(matrix, K, L)
                                                         ▷ Create a subarray individual for genetic algorithm
    rows, cols = len(matrix), len(matrix[0])
    top = random.randint(0, max(0, rows - K))
    left = random.randint(0, max(0, cols - L))
    return (top, left)
end function

function COMPUTE_FITNESS(individual, matrix, K, L, offset)
                                                         ▷ Compute fitness of an individual subarray
    top, left = individual
    return calculate_subarray_sum(matrix, top, left, K, L) + offset
end function
```

Algorithm 16 Genetic Algorithm with Constraint - Part 2: Supporting Functions

```
Continued from Part 1

function SELECT(population, fitnesses, num_parents)
                                                  ▷ Select parents based on fitness scores
    return random.choices(population, weights=fitnesses, k=num_parents)
end function

function MUTATE(individual, matrix, K, L)
                                                  ▷ Randomly mutate the individual's position
    rows, cols = len(matrix), len(matrix[0])
    top, left = individual
    top = random.randint(0, max(0, rows - K))
    left = random.randint(0, max(0, cols - L))
    return (top, left)
end function

function GENETIC_ALGORITHM(matrix, K, L, num_generations, population_size, num_parents)
                                                  ▷ Main genetic algorithm function
                                                  ▷ Calculate offset for positive fitness
    min_element = min(min(row) for row in matrix)
    offset = -min_element * K * L if min_element ¡ 0 else 0
                                                  ▷ Initialize population
    population = [create_individual(matrix, K, L) for _ in range(population_size)]

    for generation in range(num_generations) do
        fitnesses = [compute_fitness(ind, matrix, K, L, offset) for ind in population]
        parents = select(population, fitnesses, num_parents)
        next_population = [mutate(random.choice(parents), matrix, K, L) for _ in
range(population_size)]
        population = next_population
    end for

                                                  ▷ Find the best solution
    best_individual = max(population, key=lambda ind: compute_fitness(ind, matrix, K, L, offset))
    top, left = best_individual
    top_left = (top, left)
    bottom, right = top + K - 1, left + L - 1
    bottom_right = (bottom, right)
    max_sum = calculate_subarray_sum(matrix, top, left, K, L)
    return max_sum, top_left, bottom_right
end function
```

4.6.3. ***Complexity Analysis of Constrained GA***

Analyzing the complexity of the Constrained Genetic Algorithm, which focuses on segments of a fixed size (K × L), involves understanding both the time and space complexities associated with the algorithm. These complexities are influenced by various factors such as the population size (P), the number of generations (G), the dimensions of the subarray (K × L), and the size of the entire matrix (M × N).

Time Complexity Analysis
Factors Influencing Time Complexity.

- **Population Size** (P): The number of individuals in each generation affects the number of evaluations the algorithm must perform.
- **Number of Generations** (G): The total number of iterations the algorithm runs, affecting the total computation time.
- **Dimensions of the Subarray** ($K \times L$): The size of the subarray being evaluated in each individual, contributing to the computation time for the fitness function.
- **Matrix Dimensions** ($M \times N$): Evaluating the fitness involves iterating over the subarray within the matrix, adding to the computational cost.

Given these factors, the time complexity of the algorithm can be approximated as:

$$O(P \times G \times (M \times N + K \times L))$$

This formula accounts for the evaluation of each individual's fitness within the population for their respective $K \times L$ sub-arrays, as well as the potential for evaluating segments throughout the entire matrix.

Space Complexity Analysis
Factors Influencing Space Complexity. The space complexity of the algorithm is determined by:

- **Population Storage**: Storing P individuals, each representing a segment within the matrix.
- **Subarray Dimensions**: The storage required for representing the size $K \times L$ of the segment for each individual.
- **Matrix Size**: The space required to store the entire matrix of size $M \times N$.

Thus, the space complexity is influenced by the size of the population and the dimensions of both the matrix and the sub-arrays. It can be approximated as:

$$O(P \times (K \times L) + M \times N)$$

This formula reflects the space required to store the entire population of segments and the original matrix.

Conclusion

In summary, the Constrained Genetic Algorithm exhibits a complexity that is influenced by multiple factors, including the population size, the number of generations, and the dimensions of both the sub-arrays and the entire matrix. Understanding these complexities is crucial for assessing the algorithm's efficiency and scalability in solving the Matrix Maximum Segment Problem.

4.6.4. ***Genetic Algorithm: Unconstrained Variant***

The implementation of the Genetic Algorithm for the unconstrained variant of the Matrix Maximum Segment Problem involves several key steps, each tailored to effectively search for the optimal or near-optimal segment within a matrix. This variant does not restrict the size of the segment, allowing the algorithm to explore a wide range of possible segments within the matrix.

Initial Population. The GA starts with an initial population of randomly generated individuals. Each individual in this population represents a potential solution to the problem, encoded as a segment within the matrix. The segment is defined by the coordinates of its top-left and bottom-right corners, allowing the algorithm to explore all possible segments within the matrix. The diversity of the initial population is crucial as it provides a broad range of solutions from which the algorithm can evolve.

Fitness Evaluation. A critical step in GA is the evaluation of each individual's fitness, which directly influences the selection process. For our problem, the fitness of an individual is determined by the sum of the elements within the segment it represents. The fitness function is designed to maximize this sum, guiding the algorithm towards larger sums and, consequently, towards better solutions. Mathematically, the fitness of an individual representing a segment defined by coordinates $(i1, j1)$ and $(i2, j2)$ is calculated as:

$$\text{Fitness}(i1, j1, i2, j2) = \sum_{x=i1}^{i2} \sum_{y=j1}^{j2} A[x, y]$$

where A is the matrix.

Selection Process. After fitness evaluation, the GA selects individuals to form a mating pool for the next generation. Individuals with higher fitness have a greater chance of being selected. This process often involves methods like roulette-wheel selection, tournament selection, or elitism. The goal is to ensure that the fittest individuals have a higher probability of passing their genes to the next generation.

Genetic Operators. The genetic operators – crossover and mutation – are applied to individuals in the mating pool to generate a new generation. Crossover involves combining segments from two parent individuals to produce offspring, potentially inheriting favorable traits from each parent. Mutation introduces random variations in the offspring's genetic makeup, ensuring diversity in the population and enabling the exploration of new segments.

Algorithm 17 Non-Constrained Genetic Algorithm for Matrix Maximum Segment Problem - Part 1: Base Functions and Initialization

```
import random
random.seed(42)                                   ▷ Set a random seed for reproducibility

function MAX_SEGMENT_2D_GENETIC(matrix, population_size, num_generations)
                          ▷ Maximizes the sum of a submatrix in a 2D array using a genetic algorithm
    Parameters:
        matrix: The input 2D array represented as a list of lists.
        population_size: The size of the population in each generation.
        num_generations: The number of generations for the algorithm.
    Returns: The maximum sum and coordinates of the submatrix.

    function GENERATE_PROGRAM(matrix)
                                          ▷ Generates a random program (submatrix indices)
        rows, cols = len(matrix), len(matrix[0])
        i1, i2 = random.randint(0, rows - 1), random.randint(i1, rows - 1)
        j1, j2 = random.randint(0, cols - 1), random.randint(j1, cols - 1)
        return i1, i2, j1, j2
    end function

    function FITNESS(program, matrix)
                                                  ▷ Evaluates the fitness (sum of submatrix)
        i1, i2, j1, j2 = program
        submatrix = [row[j1:j2 + 1] for row in matrix[i1:i2 + 1]]
        return sum(sum(row) for row in submatrix)
    end function
```

Algorithm 18 Non-Constrained Genetic Algorithm for Matrix Maximum Segment Problem - Part 2: Recombination, Mutation, and Main Loop

```
function RECOMBINE(program1, program2)
                                                          ▷ Recombines two programs
    i1 = max(program1[0], program2[0])
    i2 = min(program1[1], program2[1])
    j1 = max(program1[2], program2[2])
    j2 = min(program1[3], program2[3])
    return i1 <= i2 and j1 <= j2 ? (i1, i2, j1, j2) : generate_program(matrix)
end function

function MUTATE(program, rows, cols)
                                                                ▷ Mutates a program
    i1, i2, j1, j2 = program
                                                  ▷ Random mutations with a probability
    if random.random() < 0.2 then
        i1 = random.randint(0, i2)
    end if
    if random.random() < 0.2 then
        i2 = random.randint(i1, rows - 1)
    end if
    if random.random() < 0.2 then
        j1 = random.randint(0, j2)
    end if
    if random.random() < 0.2 then
        j2 = random.randint(j1, cols - 1)
    end if
    return i1, i2, j1, j2
end function

                                                  ▷ Initialization of the population
population = [generate_program(matrix) for _ in range(population_size)]

                                                  ▷ Main loop for genetic algorithm
for generation in range(num_generations) do
                                                  ▷ Sort the population based on fitness
    population = sorted(population, key=lambda program: -fitness(program, matrix))

                                                  ▷ Select the top half of the population
    selected_programs = population[:population_size // 2]

                                          ▷ Generate new population by recombination and mutation
    new_population = selected_programs.copy()
    while len(new_population) ¡ population_size do
        parent1, parent2 = random.choices(selected_programs, k=2)
        child = recombine(parent1, parent2)
        child = mutate(child, len(matrix), len(matrix[0]))
        new_population.append(child)
    end while
    population = new_population
end for

                                          ▷ Return the program with the maximum fitness
best_program = max(population, key=lambda program: fitness(program, matrix))
i1, i2, j1, j2 = best_program
top_left = (i1, j1)
bottom_right = (i2, j2)
max_sum = fitness(best_program, matrix)
return max_sum, top_left, bottom_right
=0
```

4.6.5. ***Complexity Analysis of Unconstrained GA***

Time Complexity Analysis. The time complexity of the Genetic Algorithm without constraints is an important metric that determines the efficiency of the algorithm. It is influenced by several factors:

- **Population Size** (P): Affects the number of fitness evaluations per generation.
- **Number of Generations** (G): Impacts the total number of times the genetic operations are performed.
- **Matrix Size** ($M \times N$): Each fitness evaluation might require iterating over the entire matrix, especially when evaluating larger segments.

Consequently, the time complexity can be expressed as:

$$O(P \times G \times M \times N)$$

This reflects the overall computational effort in evaluating the entire population over multiple generations, considering the matrix dimensions.

Space Complexity Analysis. Space complexity is another critical aspect that determines the amount of memory required by the algorithm. The main factors contributing to space complexity are:

- **Population Storage**: Storing a population of P individuals, with each individual representing a segment within the matrix.
- **Matrix Storage**: The space required to store the original matrix of size $M \times N$.

Therefore, the space complexity is given by:

$$O(P \times M \times N)$$

This complexity accounts for the memory needed to store the population and the matrix.

Conclusion on the Unconstrained Variant

The implementation of the unconstrained variant of the Genetic Algorithm for the Matrix Maximum Segment Problem demonstrates its capability to efficiently search large and complex solution spaces. The algorithm's ability to navigate through various potential segments without predefined constraints allows for a comprehensive exploration of the matrix. While the time and space complexities may be significant, particularly for larger matrices, the GA's flexibility and adaptability make it a powerful tool for solving complex optimization problems like the Matrix Maximum Segment Problem. The efficient management of time and space resources is key to maximizing the effectiveness of the algorithm in real-world applications.

4.7. ***Ant Colony Optimization (ACO)***

Ant Colony Optimization[wik] (ACO) is a powerful probabilistic technique used in computing for finding optimized paths. It draws inspiration from the fascinating behavior of ants searching for food. This algorithm has been widely applied to various optimization problems and demonstrates remarkable adaptability. ACO is a bio-inspired algorithm that simulates ant foraging behavior, using pheromone trails and probabilistic path selection by artificial ants to find optimal solutions.

In the context of this project, we will implement both constrained and unconstrained variants of the Ant Colony Optimization algorithm to tackle the Matrix Maximum Segment Problem.

4.7.1. ***No Constraint Variant***

The no constraint variant of the Ant Colony algorithm simulates an unrestricted environment, allowing ants to explore various paths within the matrix without size limitations or constraints. This freedom enables the algorithm to explore a wide solution space, discovering potential optimal paths without specific boundaries.

Operational Workflow:. This algorithm simulates ant behavior to find the subarray with the maximum sum in a given two-dimensional array.

calculate_subarray_sum Function:.

- **Purpose:** Calculates the sum of elements in a specified subarray within the matrix.
- **Implementation:** Iterates through the elements from indices $i1$ to $i2$ and $j1$ to $j2$ in the matrix, summing their values.
- **Returns:** The sum of the elements in the subarray.

ant_colony_algorithm Function:.

- **Purpose:** Implements the core logic of the Ant Colony Optimization for the Matrix Maximum Segment Problem.
- **Initialization:** Creates a pheromone matrix and initializes variables for tracking the best solution and its sum.
- **Iterative Process:** For a set number of iterations, the algorithm:
  1. Deploys a number of ants (each representing a possible solution).
  2. Calculates the sum for the segment chosen by each ant.
  3. Updates the best solution if a better one is found.
  4. Updates the pheromone matrix based on the solutions found.
- **Returns:** The maximum sum found and the corresponding subarray's top-left and bottom-right indices.

initialize_pheromone_matrix Function:.

- **Purpose:** Initializes the pheromone matrix with a default value for all cells.

generate_random_solution Function:.

- **Purpose:** Generates a random subarray within the matrix as a potential solution.

update_pheromone_matrix Function:.

- **Purpose:** Updates the pheromone levels in the matrix based on the solutions found by the ants.
- **Implementation:** Adjusts pheromone levels by applying evaporation and then reinforcing paths that led to better solutions.

Algorithm 19 Non-Constrained Ant Colony Algorithm for Matrix Maximum Segment Problem

```
function AntColonyAlgorithm(matrix, numAnts, numIterations, evapRate)
    rows, cols ← GetSize(matrix)
    pheromoneMatrix ← InitializePheromoneMatrix(rows, cols)
    bestSolution ← None
    maxSum ← −∞
    for iteration ← 1 to numIterations do
        solutions ← []
        for ant ← 1 to numAnts do
            solution ← GenerateRandomSolution(rows, cols)
            currentSum ← CalculateSubarraySum(matrix, solution)
            solutions.append((solution, currentSum))
            if currentSum > maxSum then
                bestSolution ← solution
                maxSum ← currentSum
            end if
        end for
        UpdatePheromoneMatrix(pheromoneMatrix, solutions, evapRate)
    end for
    topLeft ← (bestSolution[0], bestSolution[2])
    bottomRight ← (bestSolution[1], bestSolution[3])
    return maxSum, topLeft, bottomRight
end function
function InitializePheromoneMatrix(rows, cols)
    return [[1 for _ in range cols] for _ in range rows]
end function
function GenerateRandomSolution(rows, cols)
    i1, j1 ← Random(0, rows − 1), Random(0, cols − 1)
    i2, j2 ← Random(i1, rows − 1), Random(j1, cols − 1)
    return (i1, i2, j1, j2)
end function
function UpdatePheromoneMatrix(pheromoneMatrix, solutions, evapRate)
    for i ← 0 to Length(pheromoneMatrix) − 1 do
        for j ← 0 to Length(pheromoneMatrix[i]) − 1 do
            pheromoneMatrix[i][j] ← pheromoneMatrix[i][j] × (1 − evapRate)
        end for
    end for
    for all (solution, score)insolutions do
        for i ← solution[0] to solution[1] do
            for j ← solution[2] to solution[3] do
                pheromoneMatrix[i][j] ← pheromoneMatrix[i][j] + score
            end for
        end for
    end for
end function
function CalculateSubarraySum(matrix, solution)
    i1, i2, j1, j2 ← solution
    sum ← 0
    for i ← i1 to i2 do
        for j ← j1 to j2 do
            sum ← sum + matrix[i][j]
        end for
    end for
    return sum
end function
```

4.7.2. ***Complexity Analysis of UnConstrained ACO***

Analyzing the complexity of the Ant Colony Algorithm without constraints involves evaluating the computational cost with respect to the number of ants, the number of iterations, and the size of the matrix.

T*ime Complexity*

The time complexity of the Ant Colony Algorithm without constraints can be expressed as:

$$O(N \times M \times A \times I)$$

Where: - N is the number of rows in the matrix. - M is the number of columns in the matrix. - A is the number of ants. - I is the number of iterations.

The time complexity depends on these factors. More ants and iterations increase the time required for convergence, while the size of the matrix ($N \times M$) determines the search space size.

S*pace Complexity*

The space complexity primarily depends on the data structures used to store the pheromone matrix and intermediate results. It can be considered as:

$$O(N \times M)$$

Where: - N is the number of rows in the matrix. - M is the number of columns in the matrix.

The space complexity is influenced by the size of the matrix.

In practice, the Ant Colony Algorithm's performance should be assessed considering both time and space constraints, as well as the specific problem requirements.

Summary

The Ant Colony Algorithm without constraints offers a heuristic approach to finding the maximum sum subarray within a matrix. Its time and space complexity depend on parameters such as the number of ants, the number of iterations, and the size of the matrix. Proper tuning of these parameters is essential for efficient performance on specific problem instances.

4.7.3. ***Constrained Variant***

The constrained variant of the Ant Colony Optimization (ACO) algorithm introduces specific constraints on the behavior of ants during their search for an optimal solution. In this variant, ants are limited to paths that conform to predefined dimensions $K \times L$. This constraint is particularly applicable in scenarios where solutions must adhere to fixed-size requirements, such as image processing or resource allocation problems.

O*perational Workflow*

The operational workflow of the constrained ACO algorithm is as follows:

1. **Initialization:** The algorithm begins by creating a pheromone matrix that guides the movement of ants within the matrix. This matrix serves as a form of communication among ants, influencing their path choices.
2. **Ant Initialization:** A set of ants is initialized with random starting positions within the matrix. These starting positions are carefully chosen to ensure compliance with the $K \times L$ constraint. Each ant keeps track of its position and the sum of the subarray it covers.
3. **Iterations:** The algorithm executes a predefined number of iterations, during which ants explore the matrix to find subarrays that maximize the objective function.
4. **Ant Movement:** In each iteration, ants move within the matrix while adhering to the constraint $K \times L$. They can move in four possible directions: right, left, down, or up. The choice of direction is probabilistic, influenced by both the pheromone levels on the paths and the quality of the subarray found.
5. **Subarray Sum Calculation:** At each step of their movement, ants calculate the sum of elements within the subarray defined by the $K \times L$ dimensions at their current position. This sum reflects the quality of the solution represented by that subarray.
6. **Comparison and Pheromone Update:** After each ant completes its path, the solutions they found are compared. The ant that discovered the subarray with the highest sum is identified. The pheromone levels on the paths traversed by this ant are updated to reinforce those paths, making them more attractive for future ants. This step is essential for guiding the search process towards optimal solutions.
7. **Convergence:** Over multiple iterations, the algorithm gradually converges towards the optimal or near-optimal solution. The pheromone levels and path choices of ants adapt to the quality of the solutions found, leading to improved results over time.
8. **Best Solution Identification:** Upon completing all iterations, the algorithm identifies the ant that discovered the subarray with the highest sum. The position of this ant represents the top-left corner of the best subarray, and its $K \times L$ dimensions are used to calculate the bottom-right corner of the subarray.

Algorithm 20 Ant Colony Optimization - Constrained Variant

```
function CalculateSubarraySum(matrix, top, left, K, L)
    sum ← 0
    for i ← top to top + K − 1 do
        for j ← left to left + L − 1 do
            sum ← sum + matrix[i][j]
        end for
    end for
    return sum
end function
function MoveAnt(ant, matrix, K, L)
    rows ← Length(matrix)
    cols ← Length(matrix[0])
    Choose a direction randomly from {(0, 1), (0, −1), (1, 0), (−1, 0)}
    Update ant.position ensuring it stays within matrix bounds
end function
function InitializeAnts(num_ants, matrix, K, L)
    ants ← []
    for i ← 1 to num_ants do
        position ← Random position within matrix bounds
        Create a new Ant instance and append to ants
    end for
    return ants
end function
function AntColonyOptimization(matrix, K, L, num_ants, num_iterations)
    if matrix = Noneor K > Length(matrix)or L > Length(matrix[0]) then
        return −∞, (0, 0), (0, 0)
    end if
    ants ← InitializeAnts(num_ants, matrix, K, L)
    for iteration ← 1 to num_iterations do
        for ant in ants do
            MoveAnt(ant, matrix, K, L)
            ant.subarray_sum ← CalculateSubarraySum(matrix, ant.position, K, L)
        end for
    end for
    best_ant ← Max(ants, key = λant : ant.subarray_sum)
    top_left ← best_ant.position
    bottom_right ← (best_ant.position[0] + K − 1, best_ant.position[1] + L − 1)
    return best_ant.subarray_sum, top_left, bottom_right
end function
```

4.7.4. ***Complexity Analysis of Constrained ACO***

Analyzing the complexity of the Ant Colony Optimization algorithm with constraints involves evaluating the computational cost with respect to the number of ants, the number of iterations, and the size of the matrix. Additionally, the dimensions of the subarray, K and L, add an extra layer of complexity.

T*ime Complexity*

The time complexity of the Ant Colony Optimization algorithm with constraints can be expressed as:

$$O(N \times M \times A \times I \times K \times L)$$

Where: - N is the number of rows in the matrix. - M is the number of columns in the matrix. - A is the number of ants. - I is the number of iterations. - K is the number of rows in the sub-array. - L is the number of columns in the sub-array.

The time complexity depends on these factors. More ants, more iterations, and larger sub-array dimensions (K and L) increase the time required for convergence.

Space Complexity

The space complexity primarily depends on the data structures used to store the ants and intermediate results. It can be considered as:

$$O(A \times K \times L)$$

Where: - A is the number of ants. - K is the number of rows in the subarray. - L is the number of columns in the subarray.

The space complexity is influenced by the number of ants and the dimensions of the subarray.

In practice, the Ant Colony Optimization algorithm's performance should be assessed considering both time and space constraints, as well as the specific problem requirements.

Summary

The Ant Colony Optimization algorithm with constraints offers a heuristic approach to finding the maximum sum subarray within a matrix while adhering to specified subarray dimensions (K and L). Its time and space complexity depend on parameters such as the number of ants, the number of iterations, and the size of the matrix, as well as the subarray dimensions. Proper tuning of these parameters is essential for efficient performance on specific problem instances.

Application to Matrix Maximum Segment Problem:

In the context of the Matrix Maximum Segment Problem, the constrained ACO algorithm aims to find the sub-array of fixed size K x L with the maximum sum. Ants explore various positions in the matrix, evaluating the sum of the sub-array at their current position. Over time, the pheromone trails lead to more promising areas of the matrix, converging on the sub-array that provides the maximum sum.

4.7.5. ***Study***

In this section, we present a detailed analysis of the performance of genetic algorithms and ant colony optimization algorithms for solving the maximal segment problem in 2D matrices. Our goal is to assess these algorithms in various scenarios, considering both constrained and unconstrained conditions.

Study Scenarios

We evaluated the algorithms using three matrix sizes:

- **Small Matrix:** Size 5x5.
- **Medium Matrix:** Size 10x10.
- **Large Matrix:** Size 20x20.

Each matrix was tested under two conditions:

1. **Without Constraints:** Allowing exploration of submatrices of any size.
2. **With Constraints:** Restricting to submatrices of size $K \times L$ (for our tests, $K = L = 3$).

Performance Metrics

Performance was assessed based on:

- **Execution Time:** The duration each algorithm takes to find the maximal segment.
- **Complexity:** Algorithmic complexity, influenced by factors like algorithm type, population size, and matrix dimensions.

Results

The experiments, conducted across different matrix sizes and conditions, yielded the following observations:

Genetic Algorithms:.

- **No Constraint:** Effective across all sizes, with reasonable execution times and complexities.
- **With Constraints:** Slight increases in execution times and complexities, yet efficient results.

Ant Colony Optimization Algorithms:.

- **No Constraint:** Competitive performance, especially in larger matrices, with efficient execution times.
- **With Constraints:** Balanced performance, marked by reasonable execution times and complexities.

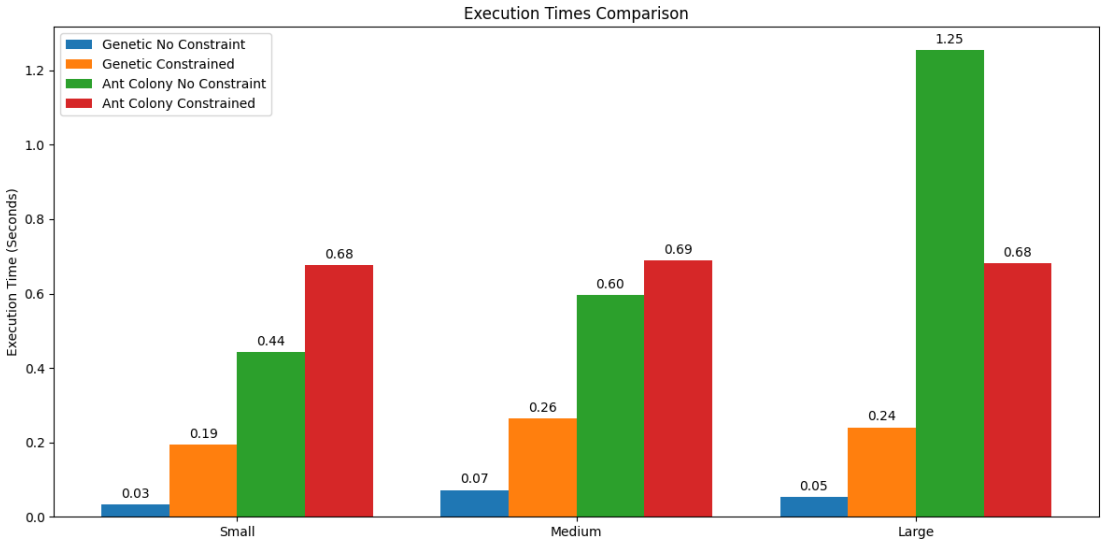


Figure 5: Algorithm Comparison in Execution Times

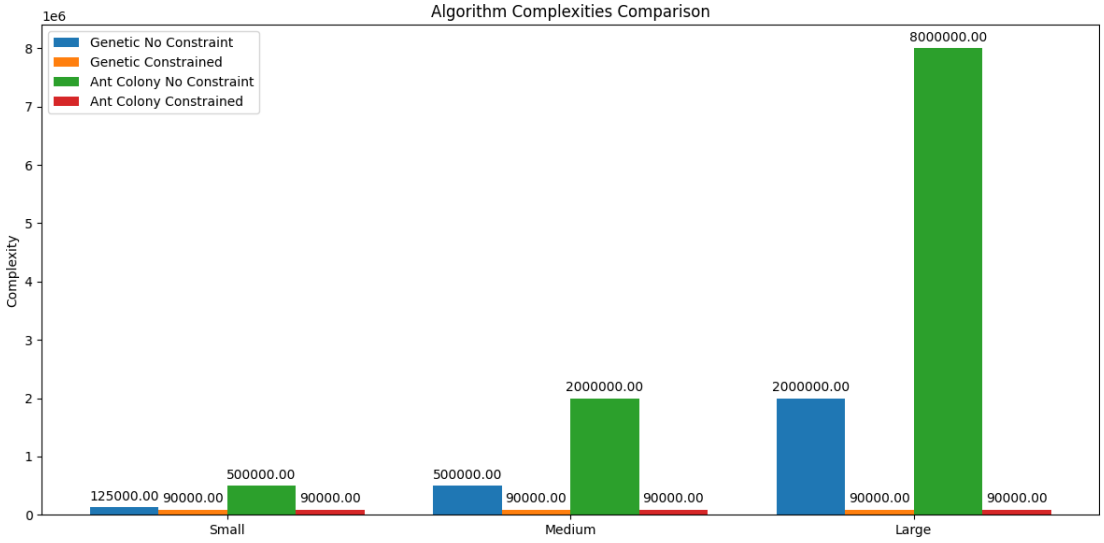


Figure 6: Comparison of Algorithm Complexities

4.7.6. ***Experimental Setup and Results***

The experimental setup encompassed varying matrix sizes and testing conditions to evaluate both genetic algorithms and ant colony optimization in solving the Matrix Maximum Segment Problem.

Testing Conditions:.

1. Without Constraints: Unrestricted submatrix exploration.
2. With Constraints: Limited to submatrices of 3x3.

Performance Metrics:.

- Execution Time (milliseconds)

- Algorithmic Complexity
- Solution Quality

4.7.7. ***Comparative Analysis and Conclusion***
Genetic Algorithms:. Efficient in smaller matrices with rapid convergence but face challenges in larger matrices due to increased complexity.

Ant Colony Optimization Algorithms:. Show better scalability with consistent high-quality solutions across different sizes, particularly effective under constraints.

Ant Colony Optimization Algorithms:.

- Small Matrix: Comparable performance to GAs but with slightly more consistent quality.
- Medium Matrix: Better handling of complexity; maintained high-quality solutions.
- Large Matrix: Demonstrated scalability with efficient time and high solution quality.
- Under Constraints: Good balance between performance and quality.

Comparative Analysis:. Genetic Algorithms excel in smaller matrices with rapid convergence but face challenges in larger matrices with increased complexity. Ant Colony Optimization Algorithms, on the other hand, show better scalability and consistently high-quality solutions across different matrix sizes, especially under constraints.

Conclusion:. Both algorithms are effective in solving the Matrix Maximum Segment Problem, with their performance and quality of solutions varying based on matrix size and constraints. Ant Colony Optimization Algorithms are generally more scalable and offer higher consistency in solution quality, making them preferable for larger matrices or constrained scenarios.

5. Summary and conclusions

Critical Analysis of Time Complexities

1. **Brute Force -** $O(M^3 \times N^3)$**:**
   The Brute force approach is highly inefficient with tremendous number of iterations for large matrices due to its cubic complexity. It lacks optimization or intelligent search, making it impractical for real-world applications with large M and N.
2. **Dynamic Programming -** $O(M^2 \times N^2)$**:**
   While still better than using brute force, the quadratic complexity of the Dynamic Programming method can cause a bottleneck when dealing with large matrices. It can have a large memory costs and indicates that some properties of the problem have not been optimized.
3. **Greedy -** $O(M \times N^2)$**:**
   Even though faster, greedy algorithm might not always yield the optimal solution. This approach might fail in cases where a global perspective of the matrix is essential.
4. **Branch and Bound 1 -** $O((M^2 \times N^2) \times \max(M, N))$**:**
   This method has an extremely high time complexity, suggesting inefficient pruning of the search space. The time complexity of this approach is the product of high complexity of dynamic programming with an additional multiplicative factor, making it impractical for even moderately sized matrices.

5. **Branch and Bound 2 - $O(4^{\max(M,N)})$:**
   The exponential complexity indicates rapid growth in computation time as matrix size increases. The method is efficient for very small matrices but quickly becomes infeasible as M and N grow.

6. **Randomized Gradient - $O(M \times N \times (M + N))$:**
   The reliance on gradients and randomness could lead to inconsistent results for this method. The quadratic factors in the time complexity shows that it can be computationally intensive for larger values of M and N.

7. **Genetic - $O(P \times G \times M \times N)$:**
   The efficiency of the approach varies parameters population size (P) and number of generations (G). The method also requires careful design of genetic processes, which can be challenging and will heavily influence the performance.

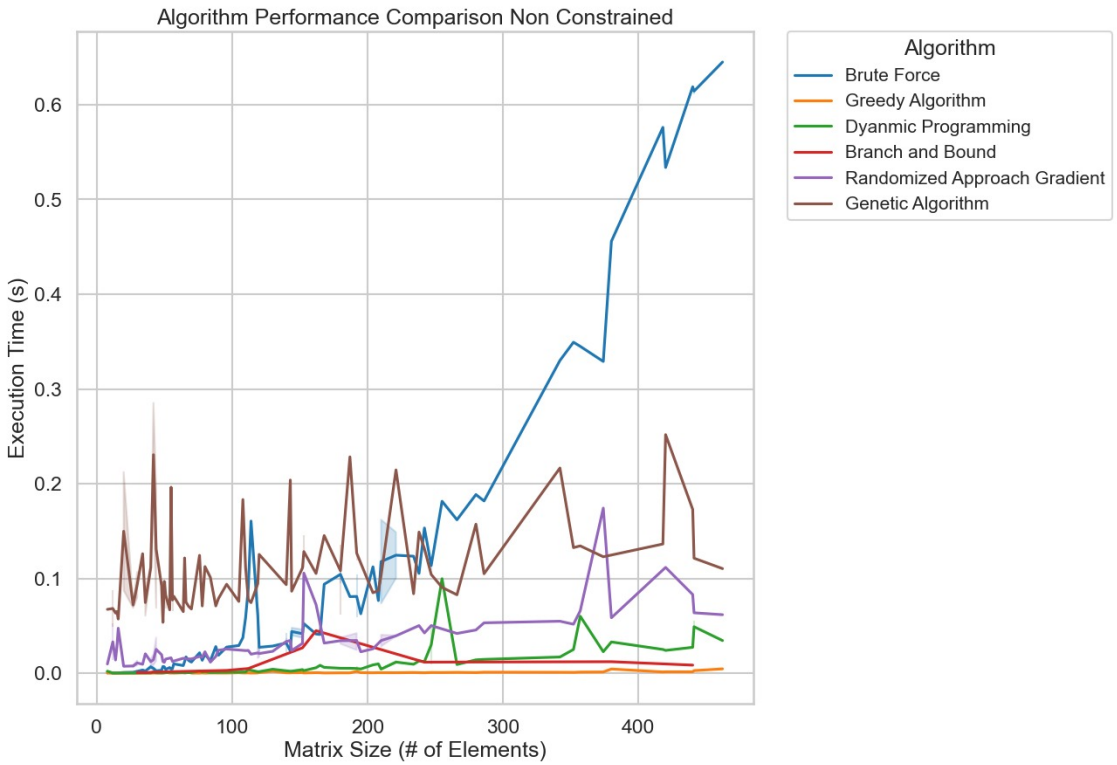


Figure 7: Algorithm Performance Comparison Non Constrained

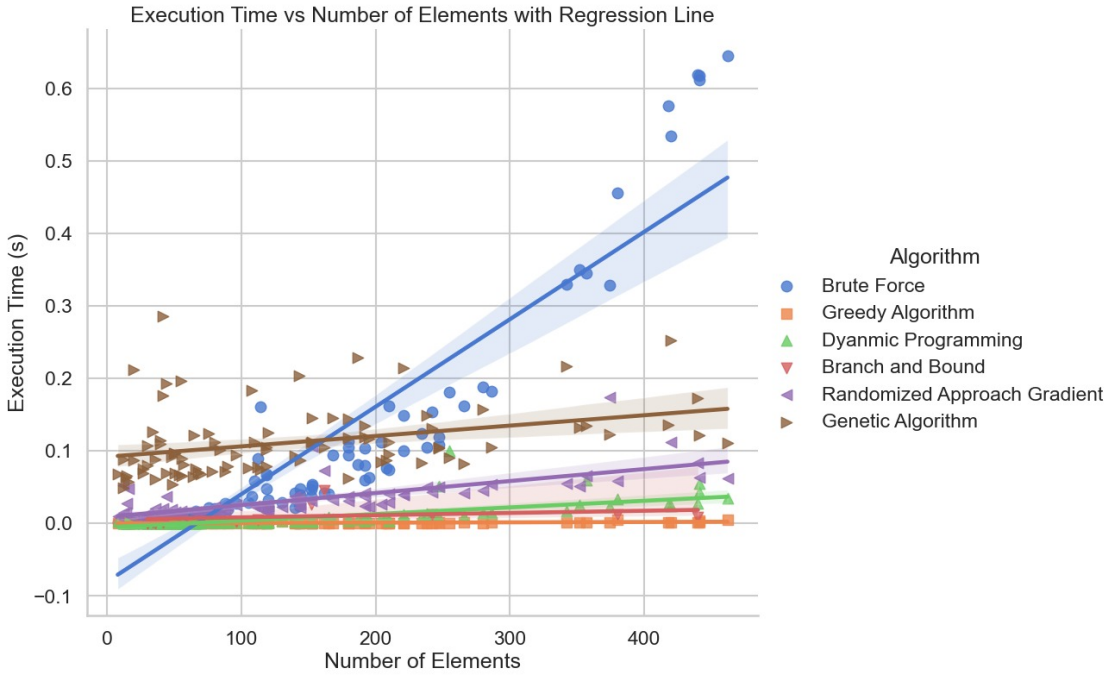


Figure 8: Algorithm Performance Comparison Non Constrained with a regression line

The execution times of several methods are shown in this graph as the size of the matrix rises. The Brute Force algorithm's execution time exhibits a clear rising trend, suggesting that higher matrix sizes negatively impact its performance. Other algorithms show reasonably steady and low execution times throughout the presented matrix sizes: the Greedy Algorithm, Dynamic Programming, Branch and Bound, and Genetic Algorithm. The graph shows that, with the exception of Brute Force, the other algorithms retain consistent execution duration as the problem size increases.

The relationship between execution time and the number of elements for a range of algorithms is shown in a scatter plot with regression lines; the steepness of each line indicates the rate at which computation time increases. The Brute Force algorithm slows down significantly with larger inputs, indicating an exponential complexity based on its steep ascent. The Greedy Algorithm, on the other hand, keeps a nearly level line, indicating that it manages growing sizes well, probably as a result of a linear or logarithmic complexity. The mild gradient of dynamic programming suggests a polynomial complexity, with the advantage of resolving overlapping sub-problems. The Branch and Bound approach performs better than Brute Force because it can eliminate less-than-optimal paths, despite having a slightly steeper slope than Dynamic Programming.Finally, because of their heuristic or random components, the Randomized Approach Gradient and Genetic Algorithm display different times; their slopes reveal broad scalability patterns but fall short of capturing the whole range of their performance attributes.

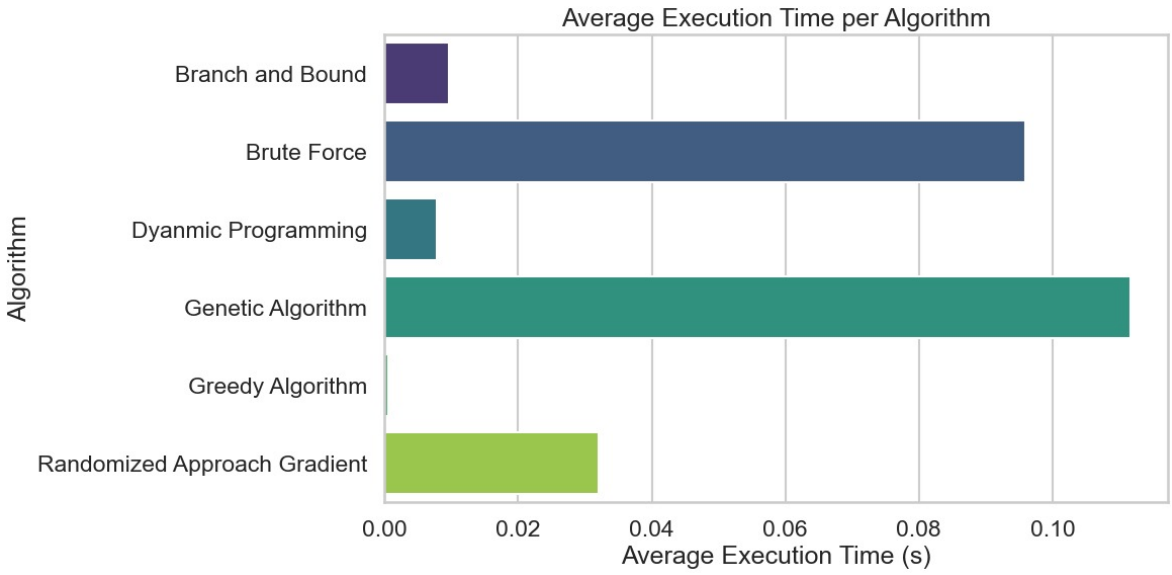


Figure 9: Bar chart for average time taken by each algorithm

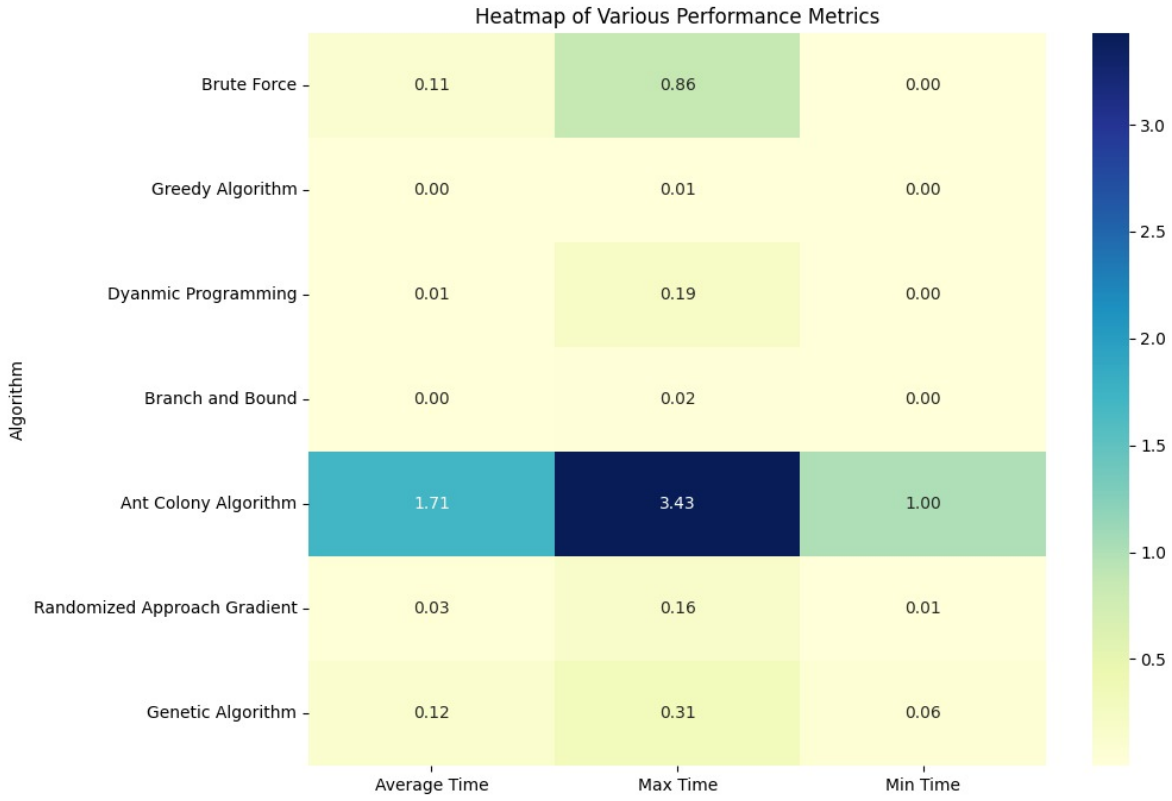


Figure 10: Heat map of average time taken by each algorithm

The heat-map shows average, maximum, and minimum execution duration along with performance characteristics for several algorithms. The Ant Colony Algorithm stands out for having the highest average and maximum times, which suggests it might not be the most effective method for the tasks at hand. The Greedy Algorithm and Branch and Bound, on the other hand, display the lowest timings across all criteria, indicating their superior efficiency. The Genetic Algorithm has more variety than the Brute Force, which has reasonable average times but different maximum times. The color intensity of the heat-map corresponds with the size of the execution time, offering a rapid visual representation of the performance of each method.

In order to scale the run time with other algorithms, the ant colony is removed from the graph. In the following graph, the Ant colony is depicted and contrasted with the genetic method

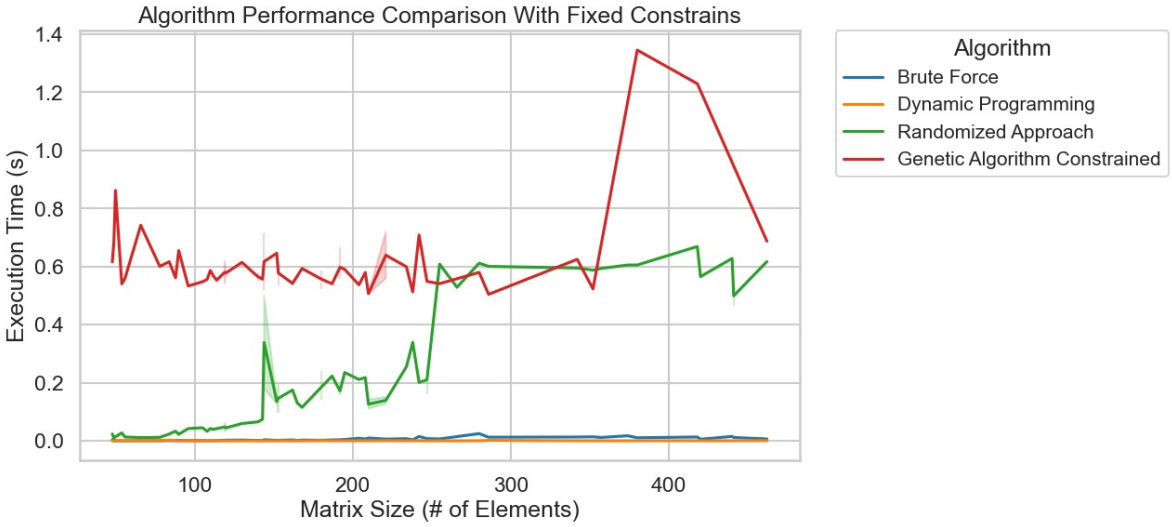


Figure 11: Algorithm Performance Comparison With Fixed Constrains

The graph is a line chart showing the execution time of four different algorithms as the matrix size increases. The Brute Force algorithm generally maintains a low execution time, while the Genetic Algorithm Constrained shows a significant spike, suggesting potential inefficiency or a specific case where it performed poorly. The Dynamic Programming and Randomized Approach remain consistent across matrix sizes, indicating stable performance.

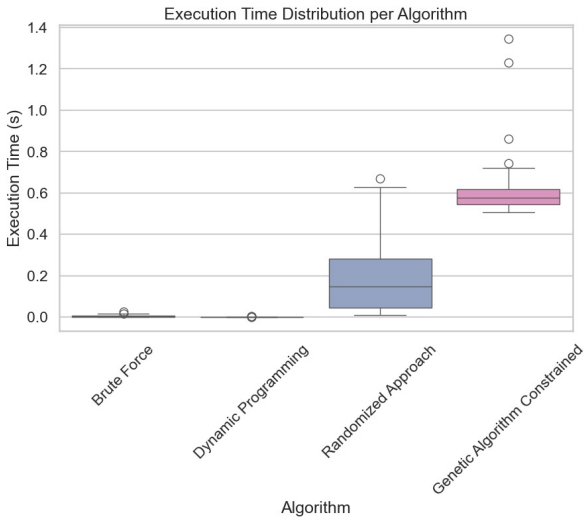


Figure 12: Execution time distribution for Constrained

The graph is a box plot that provides a distribution view of the execution times for the same algorithms. The Brute Force algorithm has the lowest median execution time but also shows outliers, indicating occasional higher execution times. The Genetic Algorithm Constrained has a higher median and larger inter-quartile range, suggesting more variability in its execution time. Dynamic Programming and the Randomized Approach show tight distributions with very few outliers, emphasizing their consistent performance.

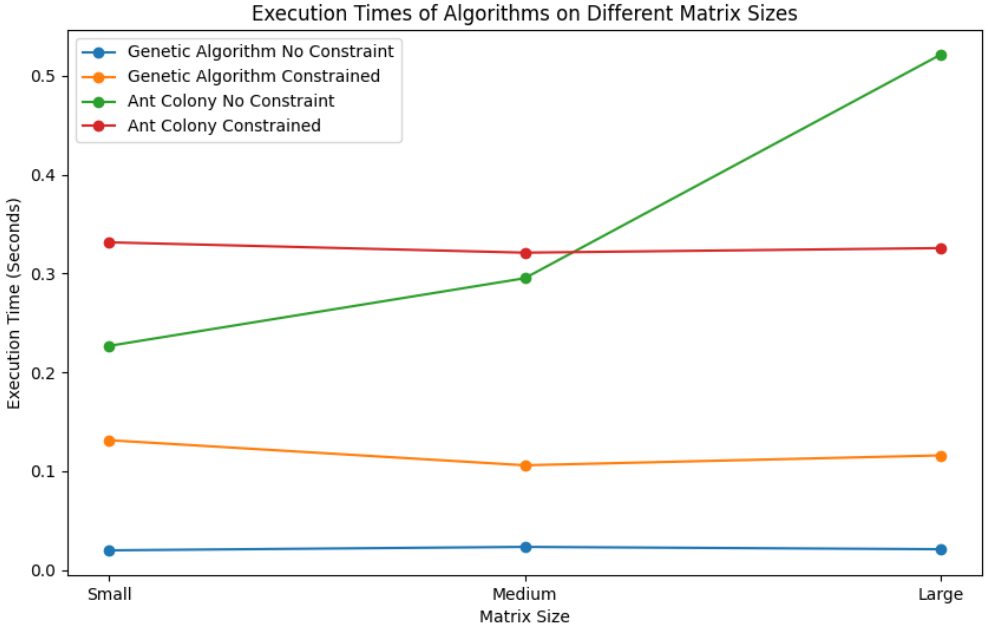


Figure 13: Algorithm Performance Comparison between Ant and genetic algorithm

The "Genetic Algorithm Constrained" seems to be an additional algorithm or a variation of the Genetic Algorithm with constraints applied. In this scenario, all the algorithms except for Brute Force have lower execution times, suggesting that the constraints help improve performance. Brute Force again shows poor scalability, while the other algorithms remain more consistent across varying problem sizes. The addition of constraints seems to have particularly benefited the Genetic Algorithm, which has a significantly reduced execution time compared to its unconstrained version in the first graph.

6. Distribution of Work

- **Meryem** :- **20% of overall work**
  - **Coding**: Two Algorithms -
    - One non-constrained version of Randomized approach.
    - One constrained version of Randomized approach.
    - Worked on generating test cases and visualization for each algorithm.
    - Worked on visualization and generating test cases for comparing all the algorithms.
    - Documentation of the concerned algorithms in GitHub.
    - Markdown and editing of the notebook that compares all algorithms.
  - **Report**:
    - Wrote the section on the Randomized approach.
    - Worked on formatting and editing the report.
    - Worked on proofreading the report and creating a glossary of definitions and acronyms.
- **Anna** :- **20% of overall work**
  - **Coding**: Three Algorithms -
    - Two non constrained versions of Branch and Bound approach.
    - One constrained version of Branch and Bound approach.
    - Documentation of the concerned algorithms in GitHub.
  - **Report**:
    - Wrote the section on Branch and Bound approach.
    - Worked on formatting and editing the report.
    - Wrote the section of work distribution and adherence to provisional planning.
- **Amgad** :- **20% of overall work**
  - **Coding**: Three Algorithms -
    - One non constrained versions of Brute Force approach.
    - One constrained versions of Brute Force approach.
    - One non constrained version of Greedy approach.
    - Created a test class to test against all approaches, test cases and visualization for each algorithm.
    - Worked on visualization for comparing all the algorithms.
    - Documentation of the concerned algorithms and the main documentation in GitHub.
  - **Report**:
    - Wrote the section on Brute force and Greedy approaches.
    - Worked on formatting and editing the report.

- **Amrithya** :- **20% of overall work**
  - **Coding**: Two Algorithms -
    - One non constrained versions of Dynamic Programming approach.
    - 1 constrained versions of Dynamic Programming approach.
    - Managed the GitHub repository.
    - Documentation of the concerned algorithms in GitHub.
  - **Report**:
    - Wrote the abstract, introduction, problem description, the section on Dynamic Programming approach.
    - Complied the bibliography and the acknowledgement parts.
    - Worked on formatting and editing the report.
- **Youssouf** :- **20% of overall work**
  - **Coding**: Four Algorithms -
    - One non constrained version of Genetic approach.
    - 1 constrained versions of Genetic approach.
    - One non constrained version of Ant-Colony approach.
    - 1 constrained versions of Ant-Colony approach.
    - Documentation of the concerned algorithms in GitHub.
  - **Report**:
    - Wrote the section on Genetic and Ant-Colony approaches.
    - Worked on formatting and editing the report.

7. Adherence to Provisional Planning

- **Week 1** : The works of the first week was according to the provisional planning and research and initial draft of the algorithm done by each member of the group.
- **Weeks 2-3** : According to the provisional plan each team member worked on their respective algorithms and completed the codes.
- **Week 4** : During the testing phase a lot of bugs were detected in the codes and debugging them didn't go according to that plan. All the debugging were done locally so as to keep the repository clean. Also one of our team member was injured during the week and couldn't complete the tasks according to the plan.
- **Week 5** : Completed the debugging of all the codes, worked on the visualization part to improve the graphs. Finished writing the report with all the algorithms and graphs.

8. Acknowledgements

We would like to express our sincere gratitude to Sri Kalidindi for their invaluable guidance and supervision throughout this project. Their expertise and insights have played a pivotal role in shaping the direction and success of our work. We would also like to extend our heartfelt thanks to Amaury Habrard for their unwavering support during this project. Furthermore, we would like to acknowledge our MLDM (Machine Learning and Data Mining) classmates who generously shared their knowledge and provided assistance when needed. We are grateful to all individuals who have contributed to this endeavor, and their support has been invaluable.

9. Checklist: Questions and Answers

This section addresses each item on the checklist through a question-and-answer format to ensure comprehensive coverage and quality of the report.

1. **Did you proofread your report?**
   Yes, the report has been thoroughly proofread to ensure clarity, coherence, and correctness.
2. **Did you present the global objective of your work?**
   The report begins with a clear statement of the global objective, which is the implementation and performance study of algorithms for the Matrix Maximum Segment Problem.
3. **Did you present the principles of all the methods/algorithms used in your project?**
   Each algorithm, including brute-force, branch-and-bound, greedy, dynamic programming, randomized approaches, and genetic programming/ant colony optimization, is explained in detail regarding its principles and application.
4. **Did you cite correctly the references to the methods/algorithms that are not from your own?**
   All external methods and algorithms are appropriately cited, adhering to academic standards.
5. **Did you include all the details of your experimental setup to reproduce the experimental results, and explain the choice of the parameters taken?**
   Comprehensive details of the experimental setup, including parameter selection and algorithm configurations, are provided to facilitate reproducibility.
6. **Did you provide curves, numerical results, and error bars when results are run multiple times?**
   The results section includes graphs, numerical data, and error analysis for experiments conducted multiple times to illustrate reliability and variance.
7. **Did you comment and interpret the different results presented?**
   Each result is accompanied by a detailed commentary and interpretation, highlighting key findings and insights.
8. **Did you include all the data, code, installation, and running instructions needed to reproduce the results?**
   The appendices contain all necessary data, code, and instructions for replicating the experiments and results.
9. **Did you engineer the code of all the programs in a unified way to facilitate the addition of new methods/techniques and debugging?**
   The code is structured consistently, with a main file for each algorithm that contains the functions that are called. To maintain consistency for testing, all functions return the same values, which are the maximum sum of the sub-matrix and the common indices of the sub-matrix and the outputs are verified in a similar way for all algorithms. It makes it simple to incorporate new techniques and streamline debugging.
10. **Did you make sure that the results of different experiments and programs are comparable?**
    Standardized testing conditions and metrics were used to ensure comparability across different experiments and programs.
11. **Did you comment sufficiently your code?**
    The provided code is extensively commented to enhance understandability and maintainability.All the files are commented properly to increase readability of the program and the Readme file of each algorithm gives the structure of input and output data
12. **Did you add a thorough documentation on the code provided?**
    The Readme file in the repository gives complete comprehensive documentation, detailing the functionality and usage of the code.
13. **Did you provide the additional planning and the final planning in the report and discuss organization aspects in your work?**

Both the initial and final project plans are included, discussing the organizational strategy and timeline of the work.

14. **Did you provide the workload percentage between the members of the group in the report?**
    The report contains a breakdown of the workload distribution among group members, ensuring transparency and accountability.
15. **Did you send the work in time?**
    The project was completed and submitted within the stipulated time frame.

References

[BSD18] Vincent Branders, Pierre Schaus, and Pierre Dupont. "Combinatorial Optimization Algorithms to Mine a Sub-Matrix of Maximal Sum". In: Jan. 2018, pp. 65–79. ISBN: 978-3-319-78679-7. DOI: 10.1007/978-3-319-78680-3_5.

[ga] ga. *Genetic algorithms*. URL: https://en.wikipedia.org/wiki/Genetic_algorithm.

[gee] geeksforgeeks. *Brute Force*. URL: https://www.geeksforgeeks.org/brute-force-approach-and-its-pros-and-cons/.

[kad] kadane. *kadane*. URL: https://en.wikipedia.org/wiki/Maximum_subarray_problem#Kadane's_algorithm.

[Kar91] Richard M. Karp. "An introduction to randomized algorithms". In: *Discrete Applied Mathematics* 34.1 (1991), pp. 165–201. ISSN: 0166-218X. DOI: https://doi.org/10.1016/0166-218X(91)90086-C. URL: https://www.sciencedirect.com/science/article/pii/0166218X9190086C.

[ml-] ml-cheatsheet.readthedocs.io. *Gradient Descent*. URL: https://ml-cheatsheet.readthedocs.io/en/latest/gradient_descent.html.

[Mor+16] David R. Morrison et al. "Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning". In: *Discrete Optimization* 19 (2016), pp. 79–102. ISSN: 1572-5286. DOI: https://doi.org/10.1016/j.disopt.2016.01.005. URL: https://www.sciencedirect.com/science/article/pii/S1572528616000062.

[Ped] DCC/FCUP Pedro Ribeiro - CRACS INESC-TEC. *Prefix Sums (2D) e Inclusion-Exclusion Principle*. URL: https://www.dcc.fc.up.pt/~pribeiro/aulas/pc1920/material/prefixsums.html.

[tan] taninamdar. *Number of Submatrices of a Matrix*. URL: https://taninamdar.files.wordpress.com/2013/11/submatrices3.pdf.

[Wik] Wikipiedia. *Maximum subarray problem*. URL: https://en.wikipedia.org/wiki/Maximum_subarray_problem.

[wik] wiki. *Ant colony optimization algorithms*. URL: https://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms.