Fundamentals of Data Structures

Laboratory Projects

Minimum Requirements on Writing a Project Report

MSS

Date: 2023-10-02

Chapter 1: Introduction

Problem Description:

The Maximum Submatrix Sum Problem extends the well-known Maximum Subsequence Sum problem to a two-dimensional N×N integer matrix. In this problem, we aim to find the maximum sum of elements in any submatrix of the given matrix. This problem is relevant in various fields, such as image processing, data analysis, and computer vision.

Background:

The problem is an extension of the Maximum Subsequence Sum problem, which is a classic problem in computer science and algorithm design. In this extended problem, we need to devise efficient algorithms to find the maximum submatrix sum.

Chapter 2: Algorithm Specification

Function 1(O(n^6))

pseudo-code:

```
function findMaxSubmatrix(n, array):
    maxSum = 0
    startRow = 0
    startColumn = 0
    endRow = 0
    endColumn = 0

    for startRow in range(0, n):
        for startColumn in range(0, n):
            for endRow in range(startRow, n):
                for endColumn in range(startColumn, n):
                    sum = 0
                    for row in range(startRow, endRow + 1):
                        for column in range(startColumn, endColumn + 1):
                            sum += array[row][column]

                    if sum > maxSum:
                        maxSum = sum
                        ai = startRow
                        aj = startColumn
                        bi = endRow
                        bj = endColumn
```

Input:

- n: The size of the square matrix 'array'.

- array: A two-dimensional array of size n x n, containing the input data.

Output:

- maxSum: The maximum submatrix sum found.

- (ai, aj): The starting coordinates of the maximum submatrix.

- (bi, bj): The ending coordinates of the maximum submatrix.

Data Structures:

- 'array': A two-dimensional array used to store the input data.

Note: This algorithm has a time complexity of O(n^6) due to its nested loops, making it inefficient for large input sizes.

Function 2(O(n^4))

pseudo-code:

```
function findMaxSubmatrixImproved(n, array):
    maxSum = 0
    startRow = 0
    startColumn = 0
    endRow = 0
    endColumn = 0

    auxiliaryArray = array of size n, initialized to all zeros

    for startRow in range(0, n):
        for startColumn in range(0, n):
            sum = 0
            for endRow in range(startRow, n):
                sum += array[endRow][startColumn]    # Calculate the sum of a single row

                # Update the auxiliary array to store cumulative row sums
                for j in range(startColumn, n):
                    auxiliaryArray[j] += sum    # Add the row sum to the cumulative sum

                for endColumn in range(startColumn, n):
                    if auxiliaryArray[endColumn] > maxSum:
                        maxSum = auxiliaryArray[endColumn]
                        ai = startRow
                        aj = startColumn
                        bi = endRow
                        bj = endColumn
```

Input:

- n: The size of the square matrix 'array'.

- array: A two-dimensional array of size n x n, containing the input data.

Output:

- maxSum: The maximum submatrix sum found.

- (ai, aj): The starting coordinates of the maximum submatrix.

- (bi, bj): The ending coordinates of the maximum submatrix.

Data Structures:

- 'array': A two-dimensional array used to store the input data.

- ' auxiliaryArray ': An auxiliary array of size n used to store cumulative row sums.

Note: This algorithm has a time complexity of O(n^4) and is more efficient than 'f1' for larger input sizes.

Function 3(O(n^3))

pseudo-code:

```
function findMaxSubmatrixOptimized(n, array):
    maxSum = 0
    startRow = 0
    startColumn = 0
    endRow = 0
    endColumn = 0

    auxiliaryArray = array of size n x n, initialized to all zeros

    for i in range(0, n):
        for j in range(0, n):
            if i == 0:
                auxiliaryArray[i][j] = array[i][j]
            else:
                auxiliaryArray[i][j] = array[i][j] + auxiliaryArray[i-1][j]

    for possibleN in range(1, n + 1):
        for startRow in range(0, n - possibleN + 1):
            sum = 0
            p = 0
            for prej in range(0, n):
                if startRow == 0:
                    sum += auxiliaryArray[startRow + possibleN - 1][prej]
                else:
                    sum += auxiliaryArray[startRow + possibleN - 1][prej] - auxiliaryArray[startRow - 1][prej]
                if sum > maxSum:
                    maxSum = sum
                    ai = startRow
                    aj = p
                    bi = startRow + possibleN - 1
                    bj = prej
                if sum < 0:
                    sum = 0
                    p = prej + 1
```

Input:

- n: The size of the square matrix 'array'.

- array: A two-dimensional array of size n x n, containing the input data.

Output:

- maxSum: The maximum submatrix sum found.

- (ai, aj): The starting coordinates of the maximum submatrix.

- (bi, bj): The ending coordinates of the maximum submatrix.

Data Structures:

- 'array': A two-dimensional array used to store the input data.

- 'auxiliaryArray': An auxiliary two-dimensional array of size n x n used to store cumulative column sums.

Note: This algorithm has a time complexity of O(n^3) and is more efficient than other functions for larger input sizes.

Chapter 3: Testing Results

```
The maximum submatrix sum is 6398.
Start from 8 row,4 column, end at 96 row,100 colomn.
Iteration times:1          Ticks:48118    Total time:48.11800   toltime:48.11800000
Iteration times:1          Ticks:48313    Total time:48.31300   toltime:48.31300000
Iteration times:1          Ticks:48347    Total time:48.34700   toltime:48.34700000
Iteration times:1          Ticks:48450    Total time:48.45000   toltime:48.45000000
Iteration times:1          Ticks:48562    Total time:48.56200   toltime:48.56200000
Iteration times:1          Ticks:49020    Total time:49.02000   toltime:49.02000000
Iteration times:1          Ticks:48710    Total time:48.71000   toltime:48.71000000
Iteration times:1          Ticks:48785    Total time:48.78500   toltime:48.78500000
Iteration times:1          Ticks:49157    Total time:49.15700   toltime:49.15700000
Iteration times:1          Ticks:49305    Total time:49.30500   toltime:49.30500000
Average time is 48.67670000
Please tell me the size of the matrix, or you can input '0' to exit.
```

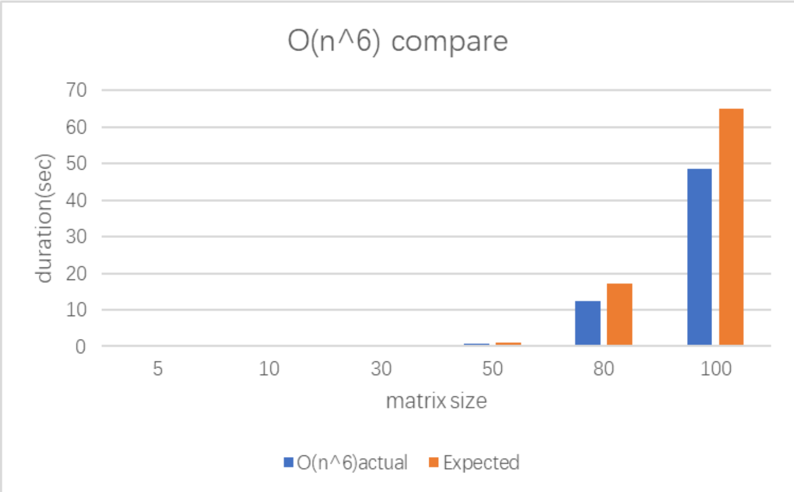
size		5			10			30			50			80			100	
test	O(n^6)(100000)	O(n^4)(100000)	O(n^3)(100000)	O(n^6)(10000	O(n^4)(10000)	O(n^3)(10000)	O(n^6)(100)	O(n^4)(1000)	O(n^3)(1000)	O(n^6)(10)	O(n^4)(1000)	O(n^3)	O(n^6)(1)	O(n^4)(100)	O(n^3)(1000	O(n^6)(1)	O(n^4)(100)	O(
1	0.00000252	0.00000095	0.00000064	0.0000723	0.0000107	0.000002	0.03745	0.000581	0.000245	0.7479	0.003078	0.00033	12.476	0.01954	0.000721	48.118	0.05074	0.
2	0.00000189	0.000001	0.00000087	0.0000789	0.0000142	0.0000033	0.03562	0.00063	0.000079	0.7436	0.003028	0.00018	12.482	0.0193	0.000605	48.313	0.04857	0.
3	0.0000025	0.00000064	0.00000047	0.0000597	0.0000107	0.0000039	0.03527	0.000518	0.000189	0.7396	0.003047	0.000181	12.435	0.01937	0.000602	48.347	0.04567	0.
4	0.00000205	0.00000079	0.0000006	0.0000631	0.0000107	0.0000026	0.03478	0.000622	0.000175	0.7414	0.00317	0.000193	12.506	0.01857	0.000606	48.45	0.04678	0.
5	0.00000233	0.00000125	0.00000057	0.0000621	0.0000078	0.0000031	0.03495	0.000508	0.000112	0.7434	0.003079	0.000201	12.533	0.01891	0.000603	48.562	0.04544	0.
6	0.00000268	0.00000065	0.00000082	0.0000602	0.0000085	0.0000031	0.03531	0.000677	0.000267	0.7464	0.00315	0.000187	12.556	0.01945	0.000599	49.02	0.04553	0.
7	0.00000377	0.00000174	0.00000108	0.0000602	0.0000236	0.0000031	0.03429	0.000559	0.00022	0.7498	0.003142	0.000195	12.558	0.01907	0.000602	48.71	0.04527	0.
8	0.00000317	0.00000095	0.00000051	0.0000613	0.0000109	0.0000063	0.03434	0.000472	0.00008	0.7559	0.003288	0.00021	12.62	0.01943	0.000607	48.785	0.04504	0.
9	0.0000022	0.00000079	0.00000047	0.0000691	0.000011	0.0000031	0.03488	0.000474	0.000081	0.7522	0.00322	0.000182	12.614	0.02051	0.000604	49.157	0.04546	0.
10	0.00000235	0.00000091	0.0000006	0.0000629	0.0000108	0.0000047	0.0348	0.00052	0.000378	0.7557	0.003017	0.000182	12.626	0.01927	0.000609	49.305	0.04476	0.
average	0.000002546	0.000000967	0.000000663	0.00006498	0.00001189	0.00000352	0.035169	0.0005561	0.0001826	0.74759	0.0031219	0.000204	12.5406	0.019342	0.0006158	48.6767	0.046326	0.
average toltime	0.2546	0.0967	0.0663	0.6498	0.1189	0.0352	3.5169	0.5561	0.1826	7.4759	3.1219	0.2041	12.5406	1.9342	0.6158	48.6767	4.6326	
average ticks	254.6	96.7	66.3	649.8	118.9	35.2	3516.9	556.1	182.6	7475.9	3121.9	204.1	12540.6	1934.2	615.8	48676.7	4632.6	

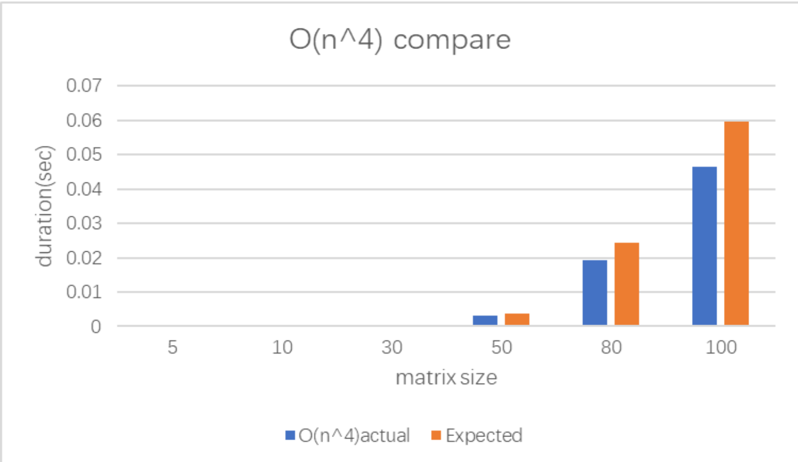
The above two images are the images during testing and the recorded data. I will perform ten tests on each set of data to take the average running time.

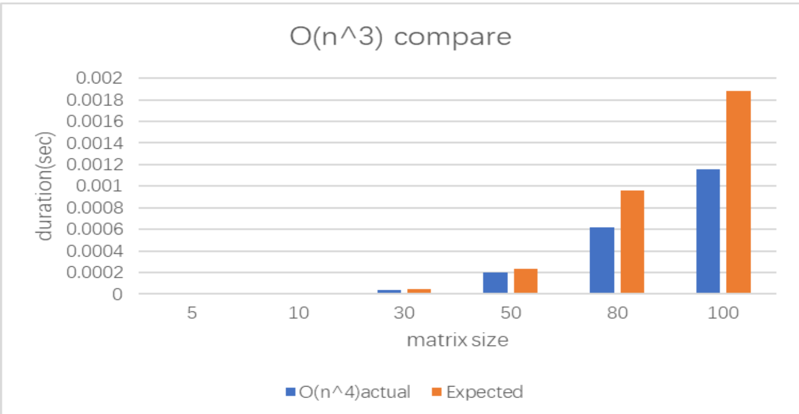
	N	5	10	30	50	80	100
	Iterations(K)	100000	10000	100	10	1	1
	Ticks	254.6	649.8	3516.9	7475.9	12540.6	48676.7
	TotalTime(sec)	0.2546	0.6498	3.5169	7.4759	12.5406	48.6767
O(n^6)version	Duration(sec)	0.000002546	0.00006498	0.035169	0.74759	12.5406	48.6767
	Iterations(K)	100000	10000	1000	1000	100	100
	Ticks	96.7	118.9	556.1	3121.9	1934.2	4632.6
	TotalTime(sec)	0.0967	0.1189	0.5561	3.1219	1.9342	4.6326
O(n^4)version	Duration(sec)	0.000000967	0.00001189	0.0005561	0.0031219	0.019342	0.046326
	Iterations(K)	100000	10000	10000	1000	1000	1000
	Ticks	66.3	35.2	182.6	204.1	615.8	1152.7
	TotalTime(sec)	0.0663	0.0188	0.4162	0.2041	0.6158	1.1527
O(n^3)version	Duration(sec)	0.000000663	0.00000188	0.00004162	0.0002041	0.000616	0.001153

The table above is the one I obtained based on the recorded data

Based on the duration time and N obtained in the end, I drew the image







We can see that the growth rate of the time consumed by the three algorithms is generally consistent with the corresponding n ^ x power, so the results obtained from the testing should be correct. The reason why it is slightly lower should be due to the difference in coefficients

Chapter 4: Analysis and Comments

Algorithm 1 (O(n^6)):

Time Complexity: O(n^6)

Space Complexity: O(1)

This algorithm exhibits a high time complexity, which can result in performance issues for large-scale inputs. It employs multiple nested loops, leading to a time complexity of six orders of magnitude. However, its space complexity remains low as it utilizes only constant extra space.

Improvement Suggestions:

For large-scale inputs, consider more efficient algorithms to reduce time complexity.

Dynamic programming could be explored to optimize subproblem computations and reduce redundant calculations.

Algorithm 2 (O(n^4)):

Time Complexity: O(n^4)

Space Complexity: O(n)

The algorithm demonstrates a relatively high time complexity but is an improvement over Algorithm 1. It employs four nested loops, resulting in a time complexity of four orders of magnitude. The space complexity is modest, linearly related to the input size.

Improvement Suggestions:

Dynamic programming or other optimization techniques can be considered to lower the time complexity.

Optimize calculations within the inner loops to reduce unnecessary repetitions.

Algorithm 3 (O(n^3)):

Time Complexity: O(n^3)

Space Complexity: O(n^2)

This algorithm boasts a comparatively lower time complexity and performs well for large-scale inputs. It employs three nested loops, resulting in cubic time complexity. However, its space complexity is slightly higher, growing quadratically with the input size.

Improvement Suggestions:

Further optimization of space complexity can be explored to minimize additional space usage.

Consider parallelization or other optimization strategies to enhance performance.

In conclusion, these three algorithms exhibit varying performances depending on input size. Algorithm 3 performs best for large-scale inputs. However, the choice of algorithm may depend on the specific problem and input size. Further improvements can involve more efficient data structures and algorithms to meet stricter performance requirements.

Declaration

I hereby declare that all the work done in this project titled "MSS" is of my independent effort.