ADVANCED ALGORITHM PROJECT 2023-2024

Hugo BRUN – Timur BALI – Marie-Louise DESSELIER – Nassim MATOUK

Table of figures

Introduction

This project brings together 7 algorithms to solve the Matrix Maximum Segment Problem in 2-Dimensional scenario.

The 7 algorithms are:

- The brute-force approach

- A Branch-and-Bound version, to be designed on your own.

- A greedy approach.

- A dynamic programming approach.

- A version based on a randomized approach.

- A version based on a genetic programming or ant colony approach.

- Another personal version.

The global objectives was to determinate which algorithm is the most efficient depending on the case.

For this project, we were 4 people from DSC:

-Hugo BRUN

-Timur BALI

-Marie-Louise DESSELIER

-Nassim MATOUK

Task sharing

We decided to split the algorithms like this:

Algorithm / task	Task Manager
Brute-force	Hugo BRUN
Branch-and-Bound	Timur BALI
Greedy approach	Marie-Louise DESSELIER
Dynamic programming	Nassim MATOUK
Randomized approach	Hugo BRUN Timur BALI
Genetic approach	Marie-Louise DESSELIER
Personal version	Timur BALI Hugo BRUN Marie-Louise DESSELIER Nassim MATOUK

Thanks to that, we have all worked on an algorithm and we worked in pairs for two of those. And we worked as a team on the personal approach.

Planning

The planning was initially this one:

Description of the algorithm

Brute-force Approach:

The Brute-force approach exhaustively tries all possible solutions and selects the best one. It systematically generates all possible combinations of solutions and evaluates each one. This approach guarantees finding the optimal solution but can be inefficient for large problem sizes.

Branch-and-Bound:

The branch and bound approach divides a problem into smaller subproblems, solves each subproblem, and uses a bounding function to avoid exploring certain branches. It require a tree structure that should avoid unnecessary exploration.

Greedy Approach:

The greedy approach makes locally optimal choices at each stage with the hope of finding a global optimum but they may not always lead to the best solution.

Dynamic Programming Approach:

The dynamic approach breaks down a problem into smaller overlapping subproblems and solves each subproblem only once, storing the solutions for future use.

Randomized Approach:

The randomized approach introduces randomness in the algorithm to achieve a probabilistic solution.

Genetic Programming or Ant Colony Approach:

Those algorithms are inspired by biological systems, these algorithms simulate natural processes like evolution or the foraging behavior of ants.

Personal Version:

This algorithm is based on searching for the largest number in the matrix, randomly selecting submatrices around it, and comparing the sum of each of those matrices.

Comparison of results and speed

We decided to compare these algorithms on the basis of their speed and results for 3 different matrices.

```python
matrice_1 = [
    [1, 2, -1, -4, -20],
    [-8, -3, 4, 2, 1],
    [3, 8, 10, 1, 3],
    [-4, -1, 1, 7, -6]
]

matrice_2 = [
    [1, 2, -1, -4, -20, 5, 3, 2, 7, 8],
    [-8, -3, 4, 2, 1, 9, 6, -2, 1, 0],
    [3, 8, 10, 1, 3, -40, 7, 5, 1, 2],
    [-4, -1, 1, 7, -6, 0, 3, -9, 6, 4],
    [2, -7, 5, 8, 2, -1, 4, -3, 2, 1],
    [-6, 3, -2, 1, 5, 7, 0, 2, -8, -3],
    [4, 2, -50, 6, 3, 1, 9, -2, 0, 8],
    [0, 1, -3, 9, 4, 2, -7, 6, 1, -5],
    [-2, 4, 6, -1, 0, 8, -3, 7, -4, 2],
    [5, 7, -8, 2, 1, -4, 6, 3, -5, -1]
]

matrice_3 = [
    [1, -20, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

Let's compare the results with the second matix, the biggest. The expected result for the second matrix is 75.

Algorithm / task	Results	Speed
Brute-force	Correct	0.0095597 seconds
Branch-and-Bound	Correct	28.2279988 seconds
Greedy approach	Correct	0.0002056 seconds
Dynamic programming	Not correct	0.0000395 seconds
Randomized approach	Not correct	0.0000392 seconds
Genetic approach	Correct	0.0964757 seconds
Personal version	Not correct	0.0034260 seconds

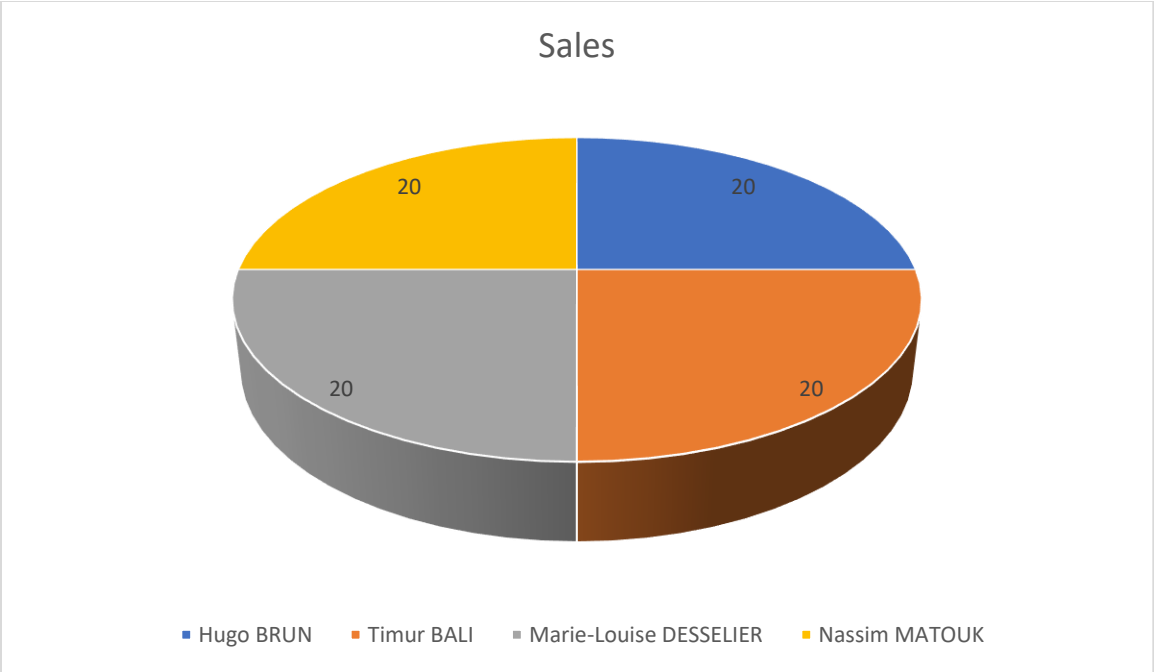
The other results can be seen at the end of the document

Conclusion

With those results we can conclude that some of those algorithms are note very efficient especially on large matrices. The Branch-and-Bound approach is efficient but also slow with a large matrix. The Greedy approach is one of the fastest and give the correct result.

So basically, the best algorithm depends on the size of the matrix, it's better to take a greedy or a brute force algorithm for big matrices.

Workload percentage



Each team member worked an average of twenty hours on the project.

Appendices

```
ster\Semestre 1\Algo\Advanced-Algo> python .\random_1.py
Somme de la sous-matrice: 18
Time to execute :  0.0000498  seconds
Sous-matrice résultante:
[1, 2, -1, -4, -20]
[-8, -3, 4, 2, 1]
[3, 8, 10, 1, 3]
[-4, -1, 1, 7, -6]
Somme de la sous-matrice: -3
Time to execute :  0.0000392  seconds
Sous-matrice résultante:
[1, 2, -1, -4, -20, 5, 3, 2, 7, 8]
[-8, -3, 4, 2, 1, 9, 6, -2, 1, 0]
[3, 8, 10, 1, 3, -40, 7, 5, 1, 2]
[-4, -1, 1, 7, -6, 0, 3, -9, 6, 4]
[2, -7, 5, 8, 2, -1, 4, -3, 2, 1]
[-6, 3, -2, 1, 5, 7, 0, 2, -8, -3]
[4, 2, -50, 6, 3, 1, 9, -2, 0, 8]
[0, 1, -3, 9, 4, 2, -7, 6, 1, -5]
[-2, 4, 6, -1, 0, 8, -3, 7, -4, 2]
[5, 7, -8, 2, 1, -4, 6, 3, -5, -1]
Somme de la sous-matrice: 9
Time to execute :  0.0000273  seconds
Sous-matrice résultante:
[1, -20, 3]
[4, 5, 6]
[7, 8, 9]
```

```
ster\Semestre 1\Algo\Advanced-Algo> python .\branch_and_bound.py
Max sum is : 29
Time to execute :  0.0033215  seconds
Biggest sub-matrix is :
[-3, 4, 2]
[8, 10, 1]
[-1, 1, 7]
Max sum is : 75
Time to execute :  28.2279988  seconds
Biggest sub-matrix is :
[8, 2, -1, 4, -3]
[1, 5, 7, 0, 2]
[6, 3, 1, 9, -2]
[9, 4, 2, -7, 6]
[-1, 0, 8, -3, 7]
[2, 1, -4, 6, 3]
Max sum is : 39
Time to execute :  0.0010377  seconds
Biggest sub-matrix is :
[4, 5, 6]
[7, 8, 9]
```

```
ster\Semestre 1\Algo\Advanced-Algo> python .\brute_force.py
Maximum Sum: 29
Time to execute :  0.0004307  seconds
[-3, 4, 2]
[8, 10, 1]
[-1, 1, 7]
Maximum Sum: 75
Time to execute :  0.0095597  seconds
[8, 2, -1, 4, -3]
[1, 5, 7, 0, 2]
[6, 3, 1, 9, -2]
[9, 4, 2, -7, 6]
[-1, 0, 8, -3, 7]
[2, 1, -4, 6, 3]
Maximum Sum: 39
Time to execute :  0.0000903  seconds
[4, 5, 6]
[7, 8, 9]
```

```
ster\Semestre 1\Algo\Advanced-Algo> python .\dynamic_approach.py
Somme maximale: 15
Time to execute :  0.0000196  seconds
Sous-matrice résultante:
[4]
[10]
[1]
Somme maximale: 35
Time to execute :  0.0000395  seconds
Sous-matrice résultante:
[2]
[1]
[7]
[8]
[1]
[6]
[9]
[-1]
[2]
Somme maximale: 18
Time to execute :  0.0000045  seconds
Sous-matrice résultante:
[3]
[6]
[9]
```

```
ster\Semestre 1\Algo\Advanced-Algo> python .\Genetic_approach.py
Somme maximale: 29
Time to execute :  0.0472070  seconds
Sous-matrice résultante:
[-3, 4, 2]
[8, 10, 1]
[-1, 1, 7]
Somme maximale: 75
Time to execute :  0.0964757  seconds
Sous-matrice résultante:
[8, 2, -1, 4, -3]
[1, 5, 7, 0, 2]
[6, 3, 1, 9, -2]
[9, 4, 2, -7, 6]
[-1, 0, 8, -3, 7]
[2, 1, -4, 6, 3]
Somme maximale: 39
Time to execute :  0.0623346  seconds
Sous-matrice résultante:
[4, 5, 6]
[7, 8, 9]
```

```
ster\Semestre 1\Algo\Advanced-Algo> python .\Greedy.py
Max sum :  29
Time to execute :  0.0000758  seconds
[-3, 4, 2]
[8, 10, 1]
[-1, 1, 7]
Max sum :  75
Time to execute :  0.0002056  seconds
[8, 2, -1, 4, -3]
[1, 5, 7, 0, 2]
[6, 3, 1, 9, -2]
[9, 4, 2, -7, 6]
[-1, 0, 8, -3, 7]
[2, 1, -4, 6, 3]
Max sum :  39
Time to execute :  0.0000466  seconds
[4, 5, 6]
[7, 8, 9]
```

```
ster\Semestre 1\Algo\Advanced-Algo> python .\Personnal_approach.py

Max subarray sum: 29
Time to execute :  0.0030055  seconds
[-3, 4, 2]
[8, 10, 1]
[-1, 1, 7]
Max subarray sum: 43
Time to execute :  0.0034260  seconds
[2, -1, -4, -20, 5, 3, 2, 7, 8]
[-3, 4, 2, 1, 9, 6, -2, 1, 0]
[8, 10, 1, 3, -40, 7, 5, 1, 2]
[-1, 1, 7, -6, 0, 3, -9, 6, 4]
[-7, 5, 8, 2, -1, 4, -3, 2, 1]
[3, -2, 1, 5, 7, 0, 2, -8, -3]
[2, -50, 6, 3, 1, 9, -2, 0, 8]
[1, -3, 9, 4, 2, -7, 6, 1, -5]
[4, 6, -1, 0, 8, -3, 7, -4, 2]
[7, -8, 2, 1, -4, 6, 3, -5, -1]
Max subarray sum: 39
Time to execute :  0.0020720  seconds
[4, 5, 6]
[7, 8, 9]
```